Vulnerabilitats de baix nivell i programari maliciós

Sergio Castillo-Pérez

PID_00178948

Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció

En l'última dècada, la seguretat computacional s'ha convertit en una necessitat omnipresent en tots els sistemes d'informació. La quantitat de vulnerabilitats que apareix cada dia no deixa immune cap sistema a no ser potencialment compromès.

En aquest context, hi ha un tipus de vulnerabilitat molt particular que es caracteritza per les greus conseqüències que pot tenir si un sistema és vulnerable. Estem parlant dels anomenats desbordaments de memòria intermèdia, o *buffers overflow*. Aquest tipus d'error no és gens nou, i la seva repercussió principal va ser causada per primera vegada ja en l'any 1988 pel cuc de Morris.

Avui dia, malgrat els avenços que s'han fet en el camp de la seguretat computacional i les tecnologies que incorporen els sistemes per a lluitar contra aquest tipus de vulnerabilitats, encara es continua descobrint programari amb aquest tipus de deficiència. El perill subjacent a aquest tipus de vulnerabilitat està en el fet que l'explotació permet, entre altres coses, l'execució de qualsevol codi arbitrari. En concret, possibilita la injecció de qualsevol codi a l'aplicació vulnerable, de manera que aquest codi és executat amb els mateixos privilegis que posseeix el programa atacat. No és estrany que per aquest motiu el programari maliciós aprofiti aquest tipus de deficiència per a prendre el control sobre aplicacions o provocar-ne la terminació.

Sens dubte, hi ha una estreta relació entre les vulnerabilitats de seguretat i el programari maliciós. Amb noves vulnerabilitats, noves vies d'explotació per al programari malintencionat. Més, si tenim en compte que la proliferació d'aquest en els últims temps també ha estat notòria. En aquest context, també hi ha hagut un canvi de paradigma radical. Mentre que en els inicis la motivació per a crear codi maliciós era el de la fama i el reconeixement, avui dia la indústria del programari maliciós es professionalitza i els seus creadors es focalitzen a obtenir beneficis econòmics. Aquest fet fa que els programadors de codi maliciós estiguin més motivats, i com a conseqüència, que l'evolució de les tècniques i la invenció de noves estratègies incorporades al programari maliciós siguin més fructíferes. Un clar exemple ha estat el cuc Stuxnet, considerat avui dia com una de les amenaces més complexes dels últims temps a causa del seu nivell de sofisticació. Sens dubte, noves bretxes s'han obert en els últims temps que seran explotades pel futur codi malintencionat, com són els dispositius mòbils, les xarxes socials, els sistemes SCADA, o les màquines virtuals.

Al llarg d'aquest mòdul estudiarem les bases teòriques dels desbordaments de memòria intermèdia, al mateix temps que s'exposaran els conceptes princi-

pals associats al codi maliciós. Analitzarem els tipus de programari maliciós existent, com és possible detectar-ne la presència, i quines estratègies usen aquests per a evadir-ne la detecció.

Objectius

Els objectius que l'estudiant ha d'haver aconseguit després d'estudiar els continguts d'aquest mòdul són els següents:

1. Aprendre les bases teòriques de les vulnerabilitats de tipus desbordament de memòria intermèdia.
2. Identificar els diferents tipus de programari maliciós en funció de les seves característiques.
3. Conèixer els mecanismes bàsics de prevenció contra el programari maliciós.
4. Tenir una visió completa de les estratègies de detecció de codi maliciós.
5. Conèixer quins són els mecanismes d'evasió utilitzats pel programari maliciós per a evitar-ne la detecció i eradicació.

1. Vulnerabilitats de baix nivell

El dia 2 de novembre de 1988 una nova amenaça va aparèixer amb el cuc de Morris. Aquest, aprofitant-se de la vulnerabilitat de diversos serveis extremadament usats com eren *fingerd*, *sendmail* i *rsh*, va causar grans danys a Internet. S'estima que el cuc va arribar a afectar un 10% de totes les màquines connectades a Internet aleshores.

Pel que sembla, i d'acord amb el seu creador, Robert Tappan Morris, la intencionalitat del cuc no era maliciosa; no obstant això, un error de programació va ser el responsable del seu comportament, que causava una denegació de servei en les màquines afectades. L'error estava en el codi responsable de la replicació, el qual provocava que una mateixa màquina pogués allotjar més d'una còpia del cuc, amb la conseqüent sobrecàrrega i alentiment del sistema per l'existència de múltiples processos creats pel mateix cuc. Sens dubte, els seus efectes van ser catastròfics per a l'època.

La propagació del cuc de Morris va ser possible gràcies a un tipus de fallada de seguretat denominat *buffer overflow* (desbordament de memòria intermèdia). Aquest tipus de vulnerabilitat permetia al cuc l'execució de codi en una altra màquina remota i vulnerable a aquesta fallada, i en creava així una nova còpia en l'altre sistema. Des de llavors, aquest tipus de vulnerabilitat s'ha convertit en un clàssic en incrementar-se la seva popularitat i en aparèixer noves tècniques d'explotació. A pesar que cada dia s'han incorporat nous mecanismes eficients de lluita contra els desbordaments de memòria intermèdia, com per exemple l'*address space layout randomization* o l'ús de l'*stack smashing protection*, cal conèixer-ne el funcionament, ja que encara estan vigents actualment.

En aquest apartat parlarem precisament d'aquest tipus de vulnerabilitats, que hem englobat sota el concepte genèric de *vulnerabilitats de baix nivell*. El motiu d'això es troba en el fet que aquest tipus de fallades es caracteritzen per produir-se des d'una perspectiva més propera al funcionament de la màquina a baix nivell.

1.1. Conceptes previs

Abans d'abordar quins són les tècniques usades pels desbordaments de memòria intermèdia, és necessari comprendre diversos aspectes de baix nivell que desenvoluparem a continuació. Atès que els mecanismes dels desbordaments de memòria intermèdia són compresos millor des d'una perspectiva de baix nivell, ens centrarem en una plataforma en concret. En particular, les explica-

cions aquí exposades estaran basades en l'arquitectura Intel x86, no obstant això, aquests mateixos conceptes poden ser extrapolats a altres arquitectures amb les particularitats que les caracteritzin.

Els aspectes que veurem a continuació es focalitzaran en l'organització de la memòria dels processos, com es fan les crides a les funcions, i com s'emmagatzemen les memòries intermèdies en memòria.

Observació

Malgrat que les nocions exposades intentaran ser autocontingudes, és recomanable que estigueu familiaritzats amb aspectes bàsics d'arquitectura de computadors i de llenguatge assemblador. A Internet es poden trobar diversos manuals, i us recomanem que n'examineu algun en cas de necessitat.

1.1.1. Organització de la memòria dels processos

Quan un programa s'executa, l'espai lineal de memòria del seu procés associat és particionat, des d'un punt de vista lògic, en diversos intervals d'adreces que anomenem **segments**. Cadascuna d'aquestes particions lògiques d'espais de memòria són utilitzades per a diferents finalitats.

Independentment de l'arquitectura i del sistema operatiu d'una màquina, en termes generals podem considerar que hi ha cinc segments (vegeu l'exemple particular de la figura 1). En particular, aquests són:

1) *Text segment*: conté el codi del procés. És conegut també amb el nom de *code segment*.

2) *Data segment*: conté les dades inicialitzades, això és, les variables estàtiques i globals els valors inicials de les quals estan emmagatzemats en l'arxiu executable, ja que el programa ha de conèixer el valor abans d'iniciar-ne l'execució.

3) *BSS segment*: conté les dades no inicialitzades, això és, totes les variables globals els valors inicials de les quals no han estat desats en l'executable, ja que el programa establirà els seus valors abans de referenciar-los.

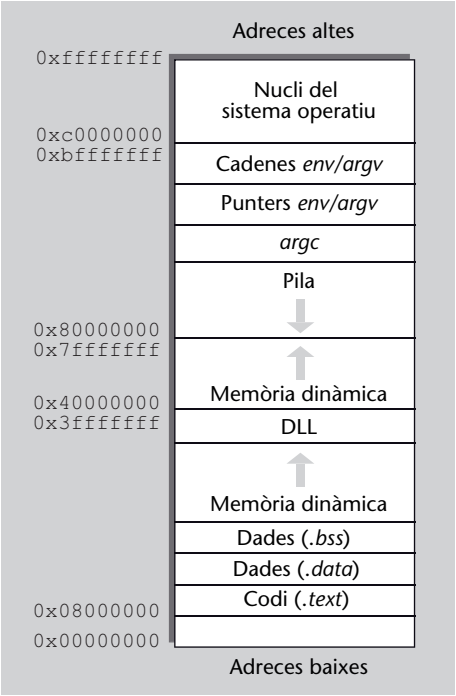
4) *Heap segment*: zona on apunten les variables que reserven memòria de manera dinàmica, i que està gestionada amb crides com `malloc`, `calloc`, o `free`, entre altres. Així mateix, aquest segment és utilitzat per a allotjar les biblioteques dinàmiques utilitzades pel procés. Aquest segment comença al final del *BBS segment* i la seva mida és dinàmica. En particular creix d'adreces baixes a adreces altes.

Operacions amb la pila

Recordeu que en una pila utilitzem dues instruccions, una per a apilar dades i una altra per a desapilar. Generalment s'utilitzen els mnemotècnics PUSH i POP, respectivament.

5) *Stack segment*: conté la pila (*stack*) del programa, una estructura de tipus LIFO, que és utilitzada principalment per a emmagatzemar les adreces de tornada quan es fan crides a funcions, als paràmetres que els passem, i a les seves variables locals. Aquest segment també és de mida dinàmica, però a diferència del *heap segment*, creix d'adreces altes a adreces baixes.

Figura 1. Organització de la memòria d'un procés en un sistema GNU/Linux



1.1.2. Crides a funcions

Una vegada entesa l'organització de la memòria d'un procés, és important considerar com una crida a una funció és feta des del punt de vista de la memòria. De manera més concreta, centrem-nos en l'ús que es fa del segment de pila des d'una perspectiva de baix nivell. Per a això, podem considerar que tota crida a una funció es pot dividir en tres passos, que s'executen cronològicament, i en cadascun d'aquests es fa un conjunt d'accions necessàries per al funcionament correcte del procés:

- **Crida**: els paràmetres de la funció són emmagatzemats en la pila i també el contingut del registre *instruction pointer**, la qual cosa permetrà determinar l'adreça de retorn després de la crida. Seguidament s'ajusta l'*instruction pointer* amb l'adreça de memòria de la funció que volem cridar, redirigint així el flux d'execució cap a aquesta.

* `EIP` **en l'arquitectura x86**

- **Pròleg**: el registre *frame pointer*** que conté l'adreça base de les variables locals és emmagatzemat en la pila i modificat perquè apunti a les variables locals de la funció que s'està cridant. Es reserva l'espai necessari en la pila per a les variables locals de la funció cridada, ajustant el registre *stack pointer**** que apunta al cim de la pila. Després del pròleg s'executarà el codi de la funció en si.

** `EBP` **en l'arquitectura x86**

*** `ESP` **en arquitectura x86**

- **Retorn (o epíleg)**: es retorna a l'estat previ de la pila abans de l'execució, restaurant el *frame pointer* i l'*instruction pointer*. La restauració de l'*instruction pointer* permet redirigir el flux d'execució al codi que va fer la crida.

Exemple

Per a comprendre-ho millor, partim d'un exemple pràctic i real d'un programa escrit en llenguatge C, que analitzarem a baix nivell després de compilar-lo. El programa en qüestió és el següent:

```
void function(int a, int b, int c) {
    char buffer1[7];
    char buffer2[9];
}

int main(void) {
    function(1,2,3);

    return 0;
}
```

Com es pot veure, el programa està compost per dues funcions simples: `main` i `function`. La funció principal `main` tan sols fa una crida a la funció `function`, i aquesta última es limita a declarar dues variables locals i a no fer res més.

Després de generar l'arxiu binari corresponent mitjançant el compilador *gcc*, si el carreguem amb el depurador *gdb* i desassemblem la funció `main` obtindrem la sortida següent:

```
0x080483bc <+0>:     push   ebp
0x080483bd <+1>:     mov    ebp,esp
0x080483bf <+3>:     sub    esp,0xc
0x080483c2 <+6>:     mov    DWORD PTR [esp+0x8],0x3
0x080483ca <+14>:    mov    DWORD PTR [esp+0x4],0x2
0x080483d2 <+22>:    mov    DWORD PTR [esp],0x1
0x080483d9 <+29>:    call   0x80483b4 <function>
0x080483de <+34>:    mov    eax,0x0
0x080483e3 <+39>:    leave
0x080483e4 <+40>:    ret
```

Fixem-nos que les tres primeres instruccions es corresponen al pròleg de la funció `main`. En concret, es desa el *frame pointer* en la pila (instrucció `push ebp`), és ajustat perquè apunti a les variables locals (instrucció `mov ebp,esp`), i es reserva espai en la pila modificant el registre `ESP` en restar-li el valor `0xc` (instrucció `sub esp,0xc`).

Les quatre instruccions següents formen part de la fase de crida a la funció `function`. De manera específica, es desen en la pila els paràmetres que es passen a la funció (valors `0x3`, `0x2` i `0x1` que apareixen en les instruccions `mov`). A continuació es desa en la pila el registre `EIP`, i se'n modifica el contingut amb l'adreça de la funció `function` per a redirigir el flux d'execució. Aquestes dues últimes accions són fetes de manera implícita per la sentència `call 0x80483b4 <function>`.

Les dues últimes instruccions es corresponen amb l'epíleg de la funció `main`. Noteu que la instrucció `leave` equival al següent:

```
mov    esp,ebp
pop    ebp
```

Per últim, la instrucció `ret` restaura el valor de l'*instruction pointer*, la qual cosa permet retornar a la pila l'estat previ a l'execució de `main`, i prosseguir amb el flux d'execució abans de la crida a `main`.

Vegem a continuació el desassemblament de la funció `function`:

```
0x080483b4 <+0>:     push   ebp
0x080483b5 <+1>:     mov    ebp,esp
0x080483b7 <+3>:     sub    esp,0x10
0x080483ba <+6>:     leave
0x080483bb <+7>:     ret
```

De nou, les tres instruccions primeres es corresponen amb el pròleg de funció `function`. Es desa el *frame pointer* en la pila i és ajustat perquè apunti a les variables locals. A continuació, es reserva espai en la pila per a les variables locals restant al registre `ESP` el valor `0x10`, la qual cosa permet allotjar les memòries intermèdies `buffer1` i `buffer2`.

Just després d'aquestes tres últimes instruccions vindria el codi propi de la funció; no obstant això, atès que aquesta no fa cap tipus d'acció, seguidament ens trobem l'epíleg.

Per tant, l'epíleg està constituït per les dues instruccions següents, les quals restauren `ESP` i `EBP` (instrucció `leave`), i també el registre `EIP` (per mitjà de la instrucció `ret`).

1.1.3. Memòria intermèdia

Una memòria intermèdia la podem definir com una quantitat de memòria reservada, limitada i contínua.

Les memòries intermèdies tenen una importància rellevant en el tipus de vulnerabilitat que ens ocupa, com veurem detingudament més endavant. De manera més precisa, un dels motius principals pel qual són possibles els desbordaments de memòria intermèdia és la inexistència de mecanismes que permetin detectar quan s'ha excedit el límit de la grandària d'una memòria intermèdia. Aquesta circumstància la trobem en determinats llenguatges de programació com són el C, el C++, o derivats. Per tant, és de vital importància que el programador extremi les mesures oportunes, quan utilitza un d'aquests llenguatges, perquè no es produeixin aquestes deficiències en el codi.

És important remarcar que no tots els llenguatges sofreixen aquest problema, i com a conseqüència no possibiliten programar codi vulnerable a desbordaments de memòria intermèdia.

En el llenguatge C, les cadenes de text són representades com un apuntador al primer byte d'una memòria intermèdia, i es considera que s'ha arribat al final d'aquestes en trobar el caràcter nul. Aquest caràcter nul està representat pel valor en hexadecimal `0x00`. Això comporta que, *a priori*, no se sàpiga la longitud de les cadenes si no es recorren fins a detectar el caràcter nul. Aquesta forma de representació pot comportar conseqüències negatives des del punt de vista de la seguretat, especialment en ser usada amb funcions denominades com a perilloses.

Exemple

Per a una comprensió millor d'aquesta problemàtica analitzem un exemple d'un programa escrit en llenguatge C:

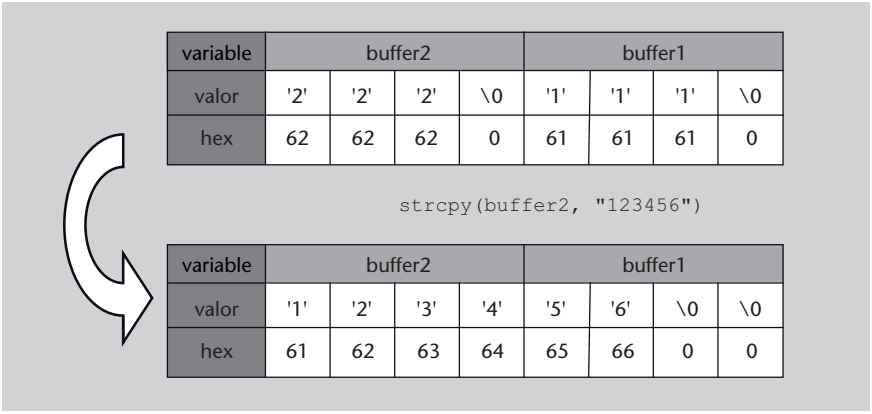
```
#include <stdio.h>
#include <string.h>

int main(void) {
    char buffer1[4] = "111";
    char buffer2[4] = "222";

    strcpy(buffer2, "123456");
    printf("%s\n", buffer1);

    return 0;
}
```

Si observem el codi veurem que el programa defineix dues memòries intermèdies denominades `buffer1` i `buffer2`. Totes dues memòries intermèdies tenen una grandària de 4 bytes, els continguts dels quals són `111` i `222`, respectivament. Noteu que en la declaració de les variables, la grandària assignada a les memòries intermèdies és de 4 bytes a causa que, al costat dels seus continguts de longitud 3, es requereix un byte addicional per al caràcter nul de final de cadena. A continuació, el programa copia –mitjançant la funció `strcpy`– la cadena de text `123456` a `buffer2`, i seguidament mostra el contingut de `buffer1`. Intuïtivament podríem cometre l'error de pensar que el resultat de l'execució d'aquest programa seria visualitzar el valor `111`, no obstant això, la realitat és una altra. Vegem què ocorre:

```
student@uoc ~ $ gcc strbof.c -o strbof
student@uoc ~ $ ./strbof
56
student@uoc ~ $
```

Com es justifica aquest resultat? N'hi ha prou d'analitzar l'evolució de la memòria per a entendre-ho. En primer lloc, recordem que les variables locals d'una funció es desen en el segment de pila. Per aquest motiu tant `buffer1` com `buffer2` –variables locals de la funció `main`– estan allotjades en el segment de pila i, a més, la reserva de memòria per a totes dues variables es fa de manera contigua. De manera gràfica podríem representar les dues variables locals tal com es mostra en la part superior de la figura 2.

Figura 2. Desbordament de memòria intermèdia amb `strcpy`



Quan s'executa la funció `strcpy`, aquesta no té en compte la grandària de la memòria intermèdia de destinació `buffer2`, i copia caràcters fins a localitzar l'identificador nul de la cadena '`123456`'. Com a conseqüència, i atès que `buffer1` i `buffer2` són contigües, el contingut de `buffer1` ha estat sobreescrit, i arriba a copiar els bytes '`56`' al costat del caràcter nul de final de cadena en l'espai reservat per `buffer1` (figura 2).

Aquesta problemàtica no és exclusiva de la funció `strcpy`, sinó que hi ha tota una llista de funcions que són susceptibles del mateix problema. Entre aquestes trobem `gets`, `strcat`, `sprintf`, o `vsprintf`, entre altres.

1.2. Desbordament de memòria intermèdia

Una vegada que s'han entès els conceptes bàsics que hem introduït en el subapartat anterior, estem en condicions d'abordar les vulnerabilitats denominades *buffers overflow*, o simplement, desbordament de memòria intermèdia.

Si analitzem les diferents variants de vulnerabilitats que apareixen sota l'epígraf de desbordaments de memòria intermèdia, ens adonarem que hi ha una gran multitud de classes. Així, una petita llista de variants que s'engloben dins d'aquest tipus de vulnerabilitat podria ser la següent:

Lectura recomanada

Sens dubte, l'article més referenciat sobre el desbordament de memòria intermèdia va ser l'escrit per Elias Levy (conegut amb el pseudònim Aleph One) el 1996, que va ser publicat en la revista *Phrack magazine* amb el títol "Smashing the Stack for Fun and Profit". Podeu trobar aquest article en aquesta adreça d'Internet: http://www.phrack.org/issues.html?id=14&issue=49

- Stack overflow
- Heap overflow
- String format
- Integer overflow
- Return into libc
- Return into environment
- Return into got

Totes les variants esmentades anteriorment es basen a explotar el mateix principi: desbordar una memòria intermèdia en usar un determinat conjunt de funcions que no inclouen cap mecanisme de control sobre les longituds.

Lectures complementàries

No s'estudiaran en profunditat les diferents variants de vulnerabilitats que apareixen sota l'epígraf de desbordaments de memòria intermèdia, però sí que es persegueix proporcionar les bases necessàries perquè pugueu acudir a altres fonts –com ara Foster i altres (2005) o Koziol i altres (2004)– i en comprengueu el funcionament. Així mateix, aquestes variants seran analitzades amb deteniment en altres assignatures, com per exemple *Programació segura*.

En el nostre cas, ens centrarem en la variant *stack overflow*, que ens servirà per a comprendre quina és la problemàtica i com és possible que aquest tipus de vulnerabilitats aconsegueixi executar codi arbitrari. Per a il·lustrar aquesta variant ho exemplificarem a partir d'un programa simple i analitzarem els aspectes importants per a comprendre-ho. El codi corresponent al programa és el següent:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[256];

    if(argc > 1) {
        strcpy(buf, argv[1]);
        printf("%s", buf);
    }

    return 0;
}
```

Com es pot observar en el codi, el programa es limita a copiar en una memòria intermèdia denominada `buff` el primer paràmetre (`argv[1]`), que es passa al programa quan s'invoca, i ho mostra tot seguit per pantalla. Aquesta acció es fa solament si al programa se li passa almenys un paràmetre, i per això la comparació que es fa amb la variable `argc`.

Aquest codi, aparentment, no té cap problema; no obstant això, què ocorreria si el primer argument del programa superés la longitud dels 256 bytes que s'han reservat per a `buf`? Vegem-ne el comportament en passar-li una cadena de 300 lletres 'A':

```
student@uoc ~ $ gcc -fno-stack-protector -z execstack -o bof bof.c
student@uoc ~ $ ./bof ‘python -c "print ’A’ * 300;" ‘
Violació de segment
student@uoc ~ $
```

Paràmetres *-fno-stack-protector -z execstack*

Els paràmetres *-fno-stack-protector -z execstack* desactiven dos mecanismes de protecció que en cas de no especificar-los no permetrien estudiar el problema dels desbordaments de memòria intermèdia.

Es produeix un error i l'execució del programa es deté sense produir-se la sortida esperada. Si executem el programa amb el mateix paràmetre però amb el depurador *gdb*, aconseguirem informació addicional d'interès:

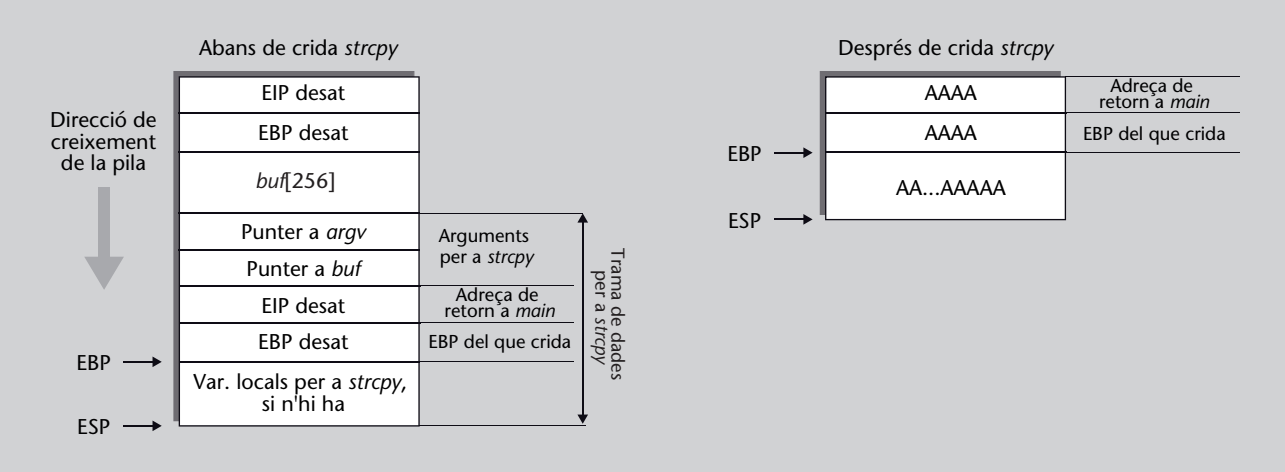
```
student@uoc ~ $ gdb ./bof
[...]
(gdb) run ‘python -c "print ’A’*300;" ‘
Starting program: /home/student/bof ‘python -c "print ’A’*300;" ‘

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info registers eip
eip            0x41414141       0x41414141
```

*** Noteu l'última línia, que mostra el contingut del registre `EIP`.**

S'ha produït una violació de segment en intentar-se executar una instrucció en l'adreça `0x41414141*`. Recordem com evoluciona la pila en cridar-se una funció i veurem on es troba la problemàtica. En primer lloc, abans de la crida a la funció `main`, s'ha desat en la pila l'adreça de tornada i també el *frame pointer* anterior. A continuació, després de la crida a la funció `main`, s'ha reservat espai per a `buf`, en tractar-se d'una variable local. Abans de la crida a la funció `strcpy`, s'han apilat en ordre invers els seus paràmetres corresponents, és a dir, el punter al primer argument del programa i el punter a `buf`. Una vegada dins de la funció `strcpy` (figura 3), aquesta còpia els 300 bytes del primer paràmetre sobre `buf`; no obstant això, si ens fixem, aquest només té reservats 256 bytes, la qual cosa provoca la sobreescriptura de l'adreça de tornada de `main`. Això comporta continuar el flux d'execució en l'adreça `0x41414141` després del retorn de `main`. De fet, es pot observar com l'adreça on es va produir la violació de segment `0x41414141`, codificada en format ASCII, és ‘`AAAA`’, que justament coincideix amb part de la cadena de text que passem al programa com a primer paràmetre.

De quina manera, llavors, es pot usar això de manera maliciosa? Bàsicament desviant el flux del programa en modificar alguna adreça de retorn per una en què l'intrús tingui un conjunt d'instruccions que li interessa executar. En el cas de la variant *stack overflow*, aquesta adreça es correspon amb alguna del rang d'adreces ocupades per la memòria intermèdia destinació d'una funció perillosa que es troba en la pila. Per a aconseguir aquesta execució, es copiarà sobre la memòria intermèdia una cadena preparada amb el conjunt d'instruccions alhora que es provoca el desbordament. Això implica que els *opcodes* de les instruccions que es volen executar no podran contenir el caràcter `NULL`, ja que en cas contrari `strcpy` (o alguna de les funcions considerades com a

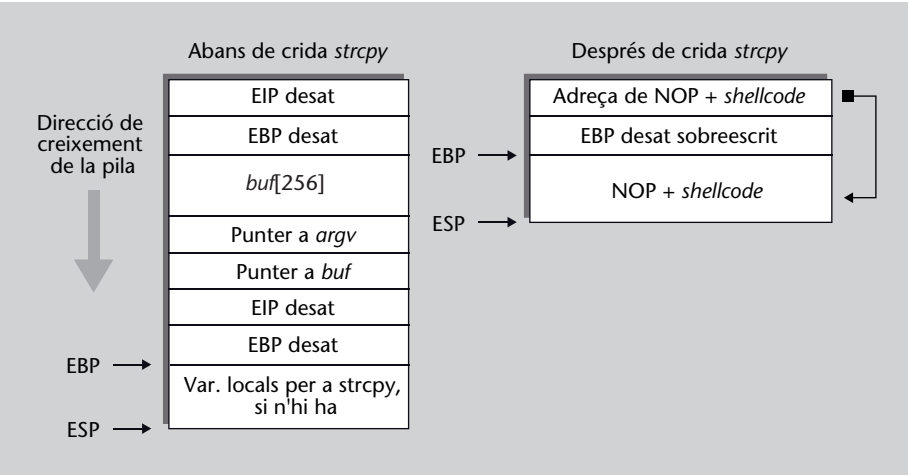
Figura 3. Evolució de la pila en un desbordament de memòria intermèdia



perilloses) no copiarà la cadena preparada íntegrament i no se sobreescriurà l'adreça de tornada.

A Internet es poden localitzar nombroses cadenes que es corresponen amb instruccions que fan determinades accions útils per a un atacant, com pot ser afegir un usuari al sistema amb una contrasenya específica, o obtenir un intèrpret d'ordres amb privilegis d'administrador. Aquestes cadenes són específiques per a cada sistema operatiu i arquitectura sobre la qual s'executen i es denominen *shellcodes*. En la figura 4 es mostra com quedaria la pila del programa exemple en passar-li com a paràmetre una cadena preparada que permet l'execució de codi arbitrari.

Figura 4. Estat de la pila en desbordar una memòria intermèdia amb un *shellcode* preparat



Una consideració que els atacants han de tenir quan preparen un *exploit* per a aquest tipus de vulnerabilitats és el fet que no es coneix l'adreça exacta de la pila on el *shellcode* serà copiat. L'intrús pot analitzar el programa vulnerable en la seva màquina per a obtenir l'adreça de retorn exacta; no obstant això, aquesta pot variar en funció de la versió del nucli del sistema operatiu o de la biblioteca del sistema *libc*. Això pot implicar que el programa que permeti aprofitar-se de la vulnerabilitat no pugui ser reutilitzat per a altres sistemes. Per

a solucionar aquesta problemàtica se solen posar davant del *shellcode* un conjunt d'instruccions (uns centenars) que no fan cap tipus d'acció, com podrien ser instruccions `NOP` (*No-Operation*). A partir d'això, l'adreça de tornada es fixa perquè caigui sobre algun lloc d'aquest conjunt d'instruccions. D'aquesta manera s'augmenten les possibilitats d'èxit independentment de la versió del nucli o de la biblioteca *libc*, i això permet que l'*exploit* pugui ser usat en diferents màquines.

2. Programari maliciós

A pesar que la idea del codi maliciós pot semblar nova, la realitat és que si volem buscar els seus orígens ens hem de remuntar a l'any 1949. En aquest any, el matemàtic John von Neumann va presentar diverses conferències a la Universitat d'Illinois sota el nom de *Theory and Organization of Complicated Automata*, en les quals englobava la teoria sobre autòmats complexos. En aquestes conferències, Neumann va establir la idea de programa emmagatzemat i va teoritzar per primera vegada la possibilitat que un programa es pogués replicar per si mateix. Sens dubte, això va representar un resultat plausible per a la teoria de la computació. Posteriorment, la seva investigació va ser publicada en l'any 1966 en el llibre titulat *Theory of self-reproducing automata*.

Durant els anys setanta van aparèixer els primers programes capaços d'autoreplicarse segons les teories que ja havia postulat John von Neumann temps enrere. Va ser més tard, l'any 1983, quan Frederick B. Cohen va encunyar el terme *virus* per a referir-se a un programa capaç d'autoreplicar-se. Un any més tard, i segons els suggeriments del seu mentor Leonard M. Adleman, Cohen va emprar el terme *virus informàtic*.

Virus informàtic segons Cohen

Cohen defineix un virus informàtic (*computer virus*, N. del T.) com un "programa que pot infectar-ne d'altres incloent-hi una còpia possiblement evolucionada de si mateix".

Cohen (1984)

Cohen va aprofundir en aquest camp, investigant sobre altres propietats dels virus informàtics, i va formalitzar la definició de virus basant-se en el model de la màquina de Turing en la seva tesi doctoral (Cohen, 1986).

En la dècada dels vuitanta, i en paral·lel als treballs de Cohen, els ordinadors personals es van popularitzar i això va representar un nou nínxol d'oportunitat per als virus informàtics. Aquesta va ser l'època d'explosió dels virus, els quals van començar a incloure en el seu codi rutines amb finalitats malicioses. Sens dubte, era el naixement del programari maliciós.

Des de la concepció de Neumann d'autòmats capaços de replicar-se, passant per la definició formal de virus informàtic introduïda per Cohen, l'evolució del codi maliciós ha estat constant i especialment de caràcter empíric. Cada dia, nous tipus de codi maliciós més complexos apareixen, fins a l'extrem que s'han hagut d'encunyar nous termes per a referir-nos a cadascuna de les no-

ves variants. En paral·lel, s'ha treballat a establir definicions formals vàlides per a qualsevol forma de programari maliciós, com la presentada per Kramer i Bradfield (2010). Sens cap dubte, aquests avenços han estat possibles gràcies als treballs previs d'investigadors com Neumann, Cohen o Adleman, les contribucions dels quals han permès establir les bases per a la lluita contra el programari maliciós actual.

Al marge de les definicions teòriques propostes per la comunitat científica per a descriure el programari maliciós, nosaltres utilitzarem una definició més pragmàtica.

El **programari maliciós** és un tipus de programari intrusiu i hostil que té com a objectiu danyar un sistema d'informació o infiltrar-s'hi sense l'aprovació ni el coneixement del seu propietari.

Origen del terme

El terme *malware* prové de la contracció de les paraules angleses *malicious* i *software*. En català utilitzem els termes *programari maliciós* o *codi maliciós* com a traducció del terme d'origen anglès.

2.1. Taxonomia del programari maliciós

Al llarg dels anys, la proliferació, la diversitat i la sofisticació del programari maliciós ha crescut de manera espectacular. Avui dia resulta complex establir una taxonomia completa, de manera que cada categoria que la compon permeti classificar el programari maliciós de manera excloent. És a dir, donat un espècimen de programari maliciós particular, és possible que aquest tingui característiques híbrides que pertanyin a més d'una de les categories en el qual el podem classificar. Així, és habitual, per exemple, trobar codi maliciós que pugui ser considerat com de propagació i ocult al mateix temps. Per tant, la taxonomia del programari maliciós que presentem aquí ha de ser entesa com una categorització genèrica, en la qual un codi maliciós donat pot pertànyer a més d'una classe de manera simultània.

En particular, considerem que hi ha tres grans categories de programari maliciós. Aquestes són: *programari maliciós de propagació automàtica*, *programari maliciós ocult* i *programari maliciós lucratiu*. Seguidament les definirem totes.

2.1.1. Programari maliciós de propagació automàtica

El programari maliciós de propagació és aquell la principal finalitat del qual és la d'estendre's de manera automàtica infectant nous sistemes d'informació.

Depenent de la forma que empra per a propagar-se distingim dues subcategories:

- **Programari maliciós de propagació per infecció vírica**. És aquell en el qual el codi maliciós es replica a si mateix en afegir-se a arxius executables, o amb la capacitat d'executar algun tipus de codi. Així mateix, aquest modifica el codi del programa original perquè, en algun instant, el flux d'execució sigui redirigit a les instruccions del programari maliciós. Cada vegada que el codi maliciós pren el control, busca nous arxius no contaminats amb la intenció d'infectar-los i, després d'això, retorna el control al programa original. A aquesta mateixa categoria, i de manera anàloga al cas anterior, pertany el programari maliciós que infecta el *master boot record* (MBR) dels discos durs, la qual cosa li permet executar-se després de les operacions pertinents de la BIOS, i prendre el control abans que es carregui el sistema operatiu. El programari maliciós que empra aquest tipus de propagació es coneix tradicionalment com a virus, i el seu nom es deu a l'analogia que tenen amb els virus biològics, i a com aquests infecten les cèl·lules d'altres organismes i es multipliquen a dins. De vegades, el terme *virus* s'utilitza de manera errònia per a referir-se a altres tipus de programari maliciós, com ara el programari espia o els troians, amb els quals no han de ser confosos, atès que aquests no infecten altres arxius.

- **Programari maliciós de propagació com a cuc**. És aquell que es replica a si mateix emprant la xarxa a la qual està connectat el sistema infectat, enviant còpies de si mateix a altres sistemes de la xarxa que no estan contaminats. Aquest enviament es fa sense la intervenció de l'usuari, i pot emprar diverses estratègies per a propagar-se, com ara l'explotació remota d'un desbordament de memòria intermèdia conegut, l'enviament indiscriminat de correus electrònics infectats amb el programari maliciós, les xarxes P2P, o els clients de missatgeria instantània, entre altres. El programari maliciós que empra aquest tipus de propagació es coneix tradicionalment com a cuc.

Origen del terme

L'origen del terme *cuc* està atribuït a John Brunner, que en la novel·la de ciència ficció *The Shockwave Rider*, publicada el 1975, descriu sota aquest nom un programa capaç de replicar-se a si mateix utilitzant una xarxa d'ordinadors.

2.1.2. Programari maliciós ocult

El **programari maliciós ocult** és un tipus de programari maliciós que es caracteritza per intentar romandre desapercebut per a l'usuari dins del sistema infectat.

Depenent de la manera com intenta passar desapercebut, i del propòsit del programari maliciós, distingim entre tres categories:

Vegeu també

En el subapartat 4.2. d'aquest mòdul s'analitzen amb més deteniment les tècniques *rootkits*.

- ***Rootkits***. Aquest tipus de codi maliciós té els seus orígens en els sistemes Unix, i la seva denominació s'utilitzava per a fer referència al conjunt d'ei-

nes que permetien a un atacant obtenir privilegis d'administrador (*root*). Avui dia, el terme *rootkit* s'empra de manera més generalitzada per a designar el conjunt de tècniques que permeten eludir la detecció i l'eliminació de qualsevol programari maliciós. Aquestes tècniques es basen en la modificació del sistema operatiu de la màquina infectada a baix nivell, per permetre l'ocultació de processos, arxius o connexions de xarxa utilitzades pel programari maliciós.

- **Troians**. El programari maliciós que pertany a aquesta categoria es caracteritza per estar emmascarat darrere d'un suposat programa legítim. La víctima, abans de la instal·lació d'aquest programari, el percep com un programa que proporciona funcionalitats del seu interès. No obstant això, després de la instal·lació, i sense el consentiment de l'usuari i sense que aquest sigui conscient, el programari maliciós actua amagant-se darrere d'un programari aparentment lícit. L'activitat oculta que du a terme el codi maliciós pot ser de diferent índole, encara que normalment sol estar enfocada al control remot de la màquina infectada, o bé al robatori d'informació. Per forçar la instal·lació del troià, l'atacant pot emprar tècniques d'enginyeria social per convèncer l'usuari, ja sigui per mitjà d'un correu o d'una pàgina web, per exemple.

Origen del terme

L'origen del terme *troià* es deu a l'analogia entre aquest tipus de programari maliciós i el cavall mitològic, segons es relata en l'*Odissea*, emprat pels grecs durant la Guerra de Troia.

- **Portes del darrere (*backdoors*)**. Aquesta categoria engloba tot programari maliciós que és instal·lat en un sistema ja compromès, i que permet eludir els mecanismes d'autenticació, alhora que roman ocult als administradors del sistema. D'aquesta manera, una porta del darrere en una màquina infectada permet a un atacant garantir l'accés al sistema en un futur d'una manera senzilla i ràpida. A pesar que l'existència de les portes del darrere no és una idea nova, la proliferació d'aquest tipus de codi maliciós ha tingut especial rellevància des de l'explosió d'Internet. La xarxa de xarxes ha permès als atacants instal·lar portes del darrere que garanteixen l'accés a sistemes compromesos de manera remota. De vegades les portes del darrere poden adoptar forma de troià i fins i tot pot incloure tècniques de *rootkits*.

2.1.3. Programari maliciós lucratiu

El programari maliciós que pertany al tipus **lucratiu** es caracteritza, com suggereix el seu nom, per proporcionar algun tipus de benefici a l'atacant.

Encara que no sempre ha de ser així, en la majoria de vegades el tipus de benefici que es persegueix és de caràcter econòmic. Depenent de la finalitat que persegueix i de com actua, distingim entre les subcategories següents:

- **Programari espia**. És un programari maliciós que registra informació sensible d'usuaris sense el seu consentiment, violant la privadesa d'aquests. La informació recollida per aquest tipus d'aplicacions pot ser de diferent índole, com per exemple dades personals, números de targeta de crèdit, hàbits de navegació web, contrasenyes, pulsacions de tecles, o fins i tot captures de pantalla. Aquesta informació es transmet a terceres parts amb finalitats com són el frau electrònic, el màrqueting per mitjà de publicitat web no consentida, o altres activitats malicioses.

Enregistradors d teclat

El terme *keylogger* s'empra per a designar el programari espia que registra les pulsacions del teclat. El mateix terme s'usa també per a fer referència a dispositius maquinari que tenen la mateixa finalitat.

- ***Ransomware***. És un tipus de programari maliciós que extorsiona els usuaris propietaris d'una màquina infectada, exigint algun tipus de pagament després de xifrar arxius, o de desactivar o bloquejar parts del sistema. Si l'usuari fa el pagament –usualment via transferència bancària o SMS amb càrrec addicional–, l'atacant proporciona algun mecanisme per a eliminar el perjudici causat pel mateix codi maliciós. En el cas particular del programari maliciós que fa ús de la criptografia per a xifrar arxius es coneix com a *criptovirus*. Aquests se solen basar en esquemes criptogràfics asimètrics o híbrids, la qual cosa impedeix l'accés al contingut dels arxius xifrats en no disposar la víctima de la clau privada.

- ***Scareware***. És un codi malintencionat que, basant-se en estratègies d'enginyeria social, pot proporcionar beneficis econòmics a l'atacant. En concret, explota l'engany, la persuasió, la coacció o la por per mitjà de missatges d'alarma o d'amenaça per forçar la víctima a fer un pagament. L'exemple més comú és el de programes que darrere de l'aparença de programari per a la detecció de programari maliciós amaguen aquest tipus de codi maliciós. Una vegada instal·lat, el programa comunica l'existència d'una quantitat elevada de programari maliciós en el sistema, quan la realitat no és així. Llavors, l'*scareware* brinda la possibilitat a l'usuari d'eliminar les amenaces detectades pagant per una versió diferent del programari de detecció. Aquest tipus de *scareware* es coneix amb el nom de *rogueware*. Altres formes de *scareware* es basen a explotar el sentiment de culpa i la por que poden sentir alguns usuaris en descarregar programari il·legal. En particular, una vegada instal·lat, aquest tipus de *scareware* mostra missatges d'advertiment indicant que s'estan violant les lleis del *copyright* i que es té identificada la IP de l'usuari. Seguidament, proposa evitar un judici fent un pagament com a manera de solucionar la situació.

- ***Bot***. És un codi maliciós que permet a un atacant controlar de manera remota la màquina que l'executa. El conjunt de màquines distribuïdes i infectades per *bots*, i controlades per un atacant, es coneixen amb el nom de *botnet* o xarxa de zombis. La suma de recursos proporcionats per cada màquina infectada permet fer activitats fraudulentes, com són atacs de denegació de servei distribuïts o l'enviament massiu de correu brossa.

Vegeu també

Per a saber més sobre les xarxes de zombis podeu consultar el mòdul didàctic "Botnets".

- **Programari de publicitat**. És un tipus de programari maliciós que de manera automàtica mostra publicitat no consentida a l'usuari, amb la finalitat

que faci algun tipus de compra. Aquesta publicitat sol aparèixer en els navegadors web com a finestres emergents, i és un comportament molest i indesitjable per a l'usuari. Alguns tipus de programari de publicitat poden ser classificats també com a programari espia, ja que la publicitat mostrada està d'acord amb informació registrada de l'activitat de l'usuari.

- ***Dialers***. És un tipus de programari malintencionat que té l'objectiu de modificar la configuració del programa de marcatge dels mòdems. En particular, modifiquen el número de telèfon per marcar per un altre la tarifa del qual és més elevada en comparació de l'associada al número legítim. Això implica un augment en l'import de la factura telefònica de l'usuari, i l'atacant n'és el beneficiari. Aquest tipus de programari maliciós va tenir èxit quan l'accés a Internet es feia amb un mòdem connectat a la xarxa de telefonia commutada. Avui dia, aquest tipus de programari està en declivi, ja que la majoria de tecnologies actuals per a l'accés a la xarxa Internet funcionen de manera diferent.

2.2. Vectors d'infecció

Un dels aspectes importants que cal tenir presents contra la lluita del programari maliciós és la identificació dels possibles vectors d'infecció. Conèixer aquestes vies d'infecció ens permet centrar els nostres esforços a dissenyar i incorporar mecanismes de seguretat adequats.

En termes generals, podem distingir dues estratègies possibles per a la infecció d'un sistema: els processos d'infecció iniciats per l'usuari víctima, i els processos d'autoinfecció iniciats per mitjà de vulnerabilitats existents en els sistemes.

En el primer cas, el programari maliciós sol estar camuflat en programes suposadament legítims i, sense el consentiment de l'usuari, s'instal·la al costat d'aquest programari. Així, programes de tipus P2P, complements per als navegadors web, programari descarregat de manera il·legal, o *cracks*, són exemples de programes que poden amagar programari maliciós. Altres vegades, la via d'infecció es basa en l'accés a una determinada web amb components ActiveX o miniaplicacions Java especialment preparades, que, després de l'autorització de l'execució per part de l'usuari, comporten la instal·lació del codi malintencionat. En qualsevol cas, el procés d'infecció requereix una aprovació d'execució tàcita per part de l'usuari. La utilització de l'enginyeria social sol estar present en aquesta metodologia, i es pot considerar fins i tot un factor decisiu en la consecució de l'èxit per als atacants. D'aquesta manera, els atacants utilitzen estratègies per a persuadir els usuaris a baixar un determinat programari, o fer determinades accions que comportin la instal·lació del programari maliciós.

En el segon cas, els mecanismes d'infecció del codi maliciós es basen en l'explotació de vulnerabilitats en el programari. Així, la visita a una determinada web preparada fent ús d'un navegador vulnerable, o l'obertura d'un arxiu especialment preparat mitjançant programari que inclogui deficiències de seguretat, poden provocar l'execució de codi arbitrari que condueixi a la instal·lació del programari maliciós. Altres vegades, un servei vulnerable ofert en una xarxa a través d'un port TCP/IP pot ser explotat pel programari maliciós per a la infecció. En qualsevol cas, en aquest procés d'infecció, la instal·lació del programari maliciós sol passar totalment desapercebuda per a l'usuari afectat, sense requerir-se cap tipus d'acció per fer que es pugui considerar motiu de sospita.

2.3. Mecanismes de prevenció

Podem considerar quatre mètodes principals per a prevenir els nostres sistemes de les infeccions de programari maliciós:

1) Foment de bones pràctiques. En primer lloc, fomentar les bones pràctiques per part dels usuaris, mantenint actualitzat tant el sistema operatiu com les aplicacions, impedir les descàrregues de programari de fonts no fiables o ignorar els correus i continguts adjunts de remitents desconeguts. Desafortunadament, aquestes pràctiques no les compleixen moltes vegades els usuaris, ni tampoc són completament efectives.

2) Disseny de patrons de protecció. Altres maneres de prevenció més tècniques es basen en el disseny de patrons de protecció. L'objectiu d'aquests patrons és impedir la infecció d'un sistema o bé reduir el dany de la infecció. Podem agrupar dins d'aquesta categoria la utilització d'anells de protecció. Aquests anells estableixen una estructura de confiança per capes en el sistema operatiu i es complementen amb maquinari específic per a poder fer una separació efectiva entre processos de confiança i processos sospitosos. Amb aquesta solució, es poden oferir diferents nivells d'accés als recursos del sistema. De fet, els anells s'organitzen de manera jeràrquica, estructurant des d'aquells dominis més privilegiats i confiables fins als de menys privilegis i nivell de confiabilitat. D'aquesta manera, es redueix el risc que processos de tipus programari maliciós ataquin el nucli del sistema operatiu (cosa que els permetria obtenir el control del sistema al complet). Aquests mètodes han demostrat que, encara que redueixen les conseqüències d'una infecció, no són totalment efectius.

3) Ús de signatures digitals. El tercer mecanisme genèric consisteix a verificar l'autenticitat del codi que s'està executant mitjançant la utilització de signatures digitals. Aquestes signatures s'associaran al codi i es verificaran abans de l'execució. Les dificultats sorgeixen aquí per a aquell codi que no hagi estat signat. En aquest cas, els usuaris hauran de decidir entre no usar les seves funcionalitats, o bé exposar-se a certs nivells de risc si aquest programa és llançat.

La pràctica ha demostrat que, davant situacions d'aquest tipus, un percentatge elevat d'usuaris prefereix executar el programari abans que verificar-ne la legítima procedència o validesa.

4) Ús d'aplicacions automàtiques. Finalment, i davant la ineficiència dels mètodes anteriors, les tendències actuals se centren en la investigació d'aplicacions automàtiques capaces de detectar i aïllar codi de tipus maliciós.

Vegeu també

Atesa l'especial rellevància que actualment tenen les aplicacions automàtiques, ens dedicarem a analitzar-ne el funcionament en l'apartat 3 d'aquest mòdul.

3. Detecció de programari maliciós

Sens dubte, la detecció del programari maliciós ha estat una de les mesures més esteses com a mecanisme de prevenció. En aquest apartat s'analitzen les tècniques més comunes emprades pel programari especialitzat en la detecció de programari maliciós. Com veurem, la complexitat d'aquestes estratègies varia. Aquesta divergència de complexitats obeeix a l'evolució que ha experimentat el codi maliciós al llarg del temps, la finalitat del qual sempre ha estat evadir els sistemes de detecció. Així, programari maliciós que ha incorporat mecanismes per a dificultar-ne la detecció ha requerit noves formes més sofisticades d'anàlisi.

Malgrat els progressos que s'han fet en el camp del programari per a la detecció de programari maliciós en els últims anys, és important remarcar que, tal com ja va postular Cohen (1987) en els seus treballs sobre els virus, el problema de la **detecció perfecta** de programari maliciós és un problema indecidible.

Detecció de programari maliciós

La detecció perfecta de programari maliciós és un problema indecidible i, per tant, no hi ha cap programa capaç de detectar tot codi maliciós possible. Així mateix, l'eradicació d'un codi maliciós en un sistema infectat no sempre està garantida, ja que depèn de la detecció.

No hi ha un programa capaç de detectar la totalitat de variants de codi maliciós que hi pot arribar a haver. Per tant, no sempre serà possible fer una detecció proactiva, la qual cosa condueix, en alguna ocasió, a una infecció inevitable dels sistemes.

No obstant això, això no significa que no puguem dissenyar mecanismes que ens permetin detectar i lluitar contra un subconjunt de la totalitat de l'espectre del programari maliciós.

Actualment, múltiples estratègies i models han estat proposats per a la detecció del programari maliciós per mitjà de programari especialitzat. Tots els models de detecció de programari maliciós poden ser classificats en dues categories en funció del tipus de detecció que fan. En particular, tenim els models *imprecisos* i els *exactes*. En el cas dels imprecisos, el motor de detecció només és capaç de determinar si un determinat objecte es tracta de programari maliciós o no, sense arribar a precisar els detalls de la versió o variant de codi maliciós de què es tracta. En contraposició, els models exactes són capaços de fer una detecció concisa, proporcionant informació particular sobre la versió de programari maliciós detectat. Evidentment, l'ús d'un model o un altre té conseqüències en el procediment d'eliminació del codi maliciós en un sistema infectat, i l'*exacte* és un procés més directe, en conèixer-se els detalls de la infecció per endavant.

Entre les diferents tecnologies per a la detecció ens centrarem en l'estudi de dues concretes, pel seu ampli ús actual. En primer lloc, analitzarem la detecció sintàctica basada en signatures, un model exacte que, per la seva naturalesa, ens impedeix lluitar contra mecanismes d'ofuscació que pot incorporar el programari maliciós. En segon lloc, veurem com la detecció semàntica es presenta com una manera de superar les deficiències de la detecció basada en signatures; no obstant això, aquest es basa en un model imprecís, amb les conseqüències que això implica.

3.1. Detecció sintàctica basada en signatures

Des d'un punt de vista històric, la detecció sintàctica basada en signatures va ser la primera tècnica emprada per a la identificació de programari maliciós. Actualment, aquesta tecnologia continua essent part del nucli dels motors de detecció de programari maliciós, i es caracteritza per la seva ràtio baixa de falsos positius.

La detecció sintàctica basada en signatures és una estratègia que es basa a localitzar dins d'objectes potencialment maliciosos (normalment fitxers o processos) algun patró que identifiqui un determinat programari maliciós conegut.

Aquests patrons –també coneguts com a *signatures sintàctiques*– estan expressats com una seqüència de bytes, i representen cadenes de text o instruccions de baix nivell en forma d'*opcodes*. Com veurem més endavant, hi ha diversos tipus de signatures amb característiques diferents.

Els motors de detecció sintàctica disposen d'una base de dades d'aquestes signatures que defineixen el conjunt de programari maliciós recognoscible. Si el motor troba alguna de les signatures de la base de dades en un objecte, aquest s'identifica de manera fefaent com un programari maliciós específic. Per a aconseguir localitzar signatures dins d'un objecte, es poden emprar diversos algorismes de cerca i concordança. Aquests algorismes estan optimitzats i tenen una complexitat fitada en funció del tipus de signatura que s'utilitza. Així mateix, aquests motors poden millorar el rendiment en limitar la cerca a parts estratègiques dels objectes, i no fer una cerca exhaustiva en tot el seu contingut.

La tecnologia de detecció de programari maliciós basat en signatures sintàctiques presenta diverses deficiències, la qual cosa el converteix en un mètode ineficaç sota certes circumstàncies:

1) Les signatures d'una base de dades sempre estan associades a programari malintencionat conegut, cosa que no permet la detecció de noves formes de

codi maliciós. Així, si un nou programari maliciós és alliberat i la seva signatura no està present en la base de dades d'un sistema, aquest podria ser infectat en passar desapercebut per al motor de detecció. De fet, una cosa tan simple com la modificació d'un codi maliciós conegut al nivell de la signatura que el detecta en permet evadir la detecció. En aquest sentit, el programari maliciós ha anat incorporant al llarg del temps estratègies d'evasió que es beneficien d'aquesta debilitat. Aquestes estratègies es basen a crear, de manera automàtica, mutacions del codi maliciós en cada nova infecció. D'aquesta manera no és possible establir patrons de detecció únics, ja que una signatura vàlida per a un programari maliciós en concret no serà útil per a la identificació de la versió mutada. De fet, s'ha demostrat com una detecció basada en signatures sintàctiques contra mutacions de tipus polimòrfic o metamòrfic condueix a un problema de tipus NP-Complet (Spinellis, 2003) o fins i tot indecidible (Filiol, 2007), respectivament.

2) La generació de les signatures pot comportar un procés llarg i tediós d'anàlisi, la qual cosa implicaria deixar exposats a una infecció els sistemes durant un temps elevat. En particular, quan es localitza un objecte que es presumeix com a maliciós, és habitual aplicar enginyeria inversa al seu codi per a determinar si es tracta o no de programari maliciós. Després d'aquesta anàlisi, i una vegada que l'objecte pot ser catalogat com a codi maliciós, es busquen característiques que l'identifiquin de manera única. A partir d'aquestes característiques d'unicitat es genera la signatura, que posteriorment es distribueix i incorpora a les bases de dades. Aquest procés, que requereix intervenció humana, no sempre és trivial, i el temps necessari es veu influenciat per les tècniques incloses en el programari maliciós per a dificultar-ne l'anàlisi. Per tant, des que el programari malintencionat s'allibera fins que les signatures són creades i incorporades a les bases de dades, hi ha un període de temps crític durant el qual no és possible la detecció per part dels motors sintàctics. Noteu l'estreta relació que hi ha entre aquest problema i el tractat en el punt anterior.

3) I finalment, a pesar que la distribució de les signatures en els millors casos es fa de manera automàtica, hi ha motors en els quals es requereix la intervenció de l'usuari per a llançar el procés d'actualització de la base de dades. Si aquest procés manual no s'efectua a temps, un sistema es pot veure compromès pel programari malintencionat, en no poder-se detectar.

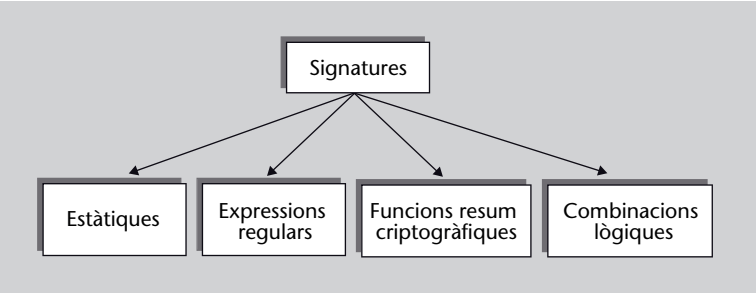
3.1.1. Tipus de signatures

En el subapartat 3.1. hem pogut veure com les signatures sintàctiques són el pilar de la metodologia de detecció que estem tractant. Amb l'objectiu de millorar la precisió en la identificació de programari maliciós, quatre tipus de signatures sintàctiques són utilitzades. De manera més concisa, aquestes són (figura 5):

1) Signatures com cadenes estàtiques. Les signatures com cadenes estàtiques són la forma més bàsica entre les diferents possibles. Aquestes estan definides

com una seqüència de bytes consecutius de longitud arbitrària. Si aquesta seqüència es localitza en un objecte analitzat, llavors és identificat com a codi maliciós.

Figura 5. Tipus de signatures sintàctiques



2) Signatures amb expressions regulars. Aquest tipus de signatures suporta expressions regulars en la seva definició. D'aquesta manera és possible localitzar concordances de bytes segons repeticions, rangs, combinacions, etc. A causa d'això, l'algorisme de cerca i concordança té més complexitat que el de les cadenes estàtiques i, per tant, menys velocitat en el procés d'identificació de programari maliciós. No obstant això, el suport d'expressions regulars el dota de més flexibilitat per a detectar programari maliciós més complex, o variants respecte a una versió de programari maliciós ja coneguda. Així, sovint, una única signatura que utilitza expressions regulars permet detectar tota una família de variants d'un determinat codi maliciós. Com és lògic pensar, aquestes signatures genèriques són útils si, en alliberar-se una variant, aquesta comparteix el mateix patró identificatiu amb versions anteriors.

3) Signatures basades en funcions resum criptogràfiques. Aquesta variant de signatures sintàctiques se sustenta en l'ús de funcions resum criptogràfiques. Recordem que aquestes funcions es caracteritzen per les propietats següents:

a) Compressió: el valor d'aplicar una funció resum a un missatge té una mida constant, i generalment inferior a la mida del missatge.

b) Unidireccionalitat: donat el resum d'aplicar una funció resum a un missatge, és impossible reconstruir el missatge original.

c) Facilitat de càlcul: donat un missatge de longitud arbitrària, és fàcil i ràpid (per a un ordinador) calcular-ne la funció resum.

d) Difusió: donat un missatge m, si modifiquem tan sols un bit de m i el denominem m', llavors el valor $hash(m')$ hauria de modificar aproximadament la meitat dels bits respecte a $hash(m)$.

e) Resistència feble a col·lisions: donat un missatge m, no és computacionalment factible trobar un altre missatge m' tal que $m \neq m'$, però que $hash(m) = hash(m')$.

f) Resistència forta a col·lisions: no és computacionalment factible trobar una parella de missatges m, m' tal que $m \neq m'$, però que $hash(m) = hash(m')$.

Aquestes propietats permeten desar les signatures com el càlcul d'una funció resum sobre un objecte (o part d'aquest) corresponent a codi maliciós. Els avantatges de l'ús d'aquest tipus de signatures és doble. En primer lloc, poden reduir l'espai necessari per a emmagatzemar una signatura, sempre que la mida resultant de calcular la funció resum sigui inferior a la mida d'un altre tipus de signatura alternativa. En segon lloc, una signatura sintàctica basada en funcions criptogràfiques pot ser computacionalment menys costosa de localitzar que els algorismes de cerca d'altres tipus de signatures.

4) Combinacions lògiques de signatures. Aquest tipus de signatures combina múltiples subsignatures sintàctiques relacionant-les mitjançant operadors lògics, cosa que permet definir patrons més flexibles i precisos. Per tant, la cerca de la nova signatura per a la identificació d'un determinat programari maliciós implicarà verificar que es compleix l'expressió lògica composta per totes les subsignatures. Aquesta estratègia pot ser utilitzada, per exemple, per a detectar una família de variants d'un codi maliciós concret. Així, es podria definir una signatura sintàctica genèrica per a totes les variants, i després una d'específica per a cada variant particular. Per a aquest escenari, combinaríem amb l'operador lògic `AND` la signatura genèrica amb la particular per a la detecció d'una variant concreta.

Cadascuna d'aquestes quatre signatures té associat un algorisme de cerca i concordança per a la identificació de codi malintencionat, i la complexitat algorítmica del qual és diferent per a cadascun.

La utilització d'un tipus o un altre de signatura varia en funció del programari maliciós que ha d'identificar i, per tant, no es pot afirmar que cap no sigui millor que una altra. Depenent del cas particular de programari malintencionat, l'elecció d'una o una altra millorarà la precisió i el temps necessari en la identificació. És habitual trobar motors de detecció sintàctica que suporten diversos tipus de signatures de manera simultània.

3.1.2. Àmbits de cerca

Amb l'objectiu de millorar el rendiment en el procés de detectar programari maliciós, cada signatura sintàctica té associat un àmbit de cerca.

Entenem per *àmbit de cerca* una restricció que limita la localització de la signatura a un tipus d'objecte o a una part del seu contingut. Això permet restringir el procés de cerca, amb l'estalvi consegüent de còmput en comparació d'una cerca exhaustiva en tots els tipus d'objecte o en la totalitat dels seus continguts.

De vegades, aquests àmbits de cerca poden ser combinats per fitar encara més la localització de les signatures. Distingim cinc maneres diferents d'especificar àmbits de cerca. Aquests són:

1) Totalitat. Aquest àmbit de cerca es refereix a la totalitat de l'objecte per analitzar. És a dir, la signatura s'intentarà localitzar en qualsevol posició dins de tot l'objecte. Per tant, des d'un punt de vista de còmput, es correspon amb el cas més costós, atès que requereix una cerca exhaustiva.

2) Desplaçament. Aquest tipus d'àmbit especifica el punt inicial de la cerca dins d'un objecte a partir d'un desplaçament. Aquest desplaçament, indicat en nombre de bytes, es pot fer respecte a l'inici de l'objecte (desplaçament positiu) o respecte al final (desplaçament negatiu). Així mateix, aquest desplaçament també es pot fer en relació amb seccions específiques i no sobre la totalitat de l'objecte. Algunes signatures també permeten definir en el seu àmbit de cerca una grandària de bytes respecte al desplaçament, la qual cosa defineix una porció fitada de l'objecte on es farà la localització.

3) Filtres d'objectes. Aquest àmbit de cerca restringeix la localització de les signatures a objectes que compleixen una sèrie de característiques, com pot ser tipus d'arxiu, grandària de l'objecte, nombre de seccions contingudes, etc.

4) Seccions específiques. El contingut dels objectes en un sistema sol estar estructurat d'una manera específica que depèn del seu tipus. Així, per exemple, l'estructura dels executables en la plataforma Windows, coneguda com a PE (*portable executable*) és diferent de la dels documents de tipus PDF (*portable document format*), o de les imatges JPG. L'organització d'aquestes estructures se sol fer en seccions o capçaleres, i cadascuna conté un tipus d'informació particular. L'àmbit de cerca basat en seccions específiques se centra precisament a restringir la cerca a alguna d'aquestes seccions. Per tant, un àmbit d'aquest tipus haurà d'especificar la secció particular de l'objecte on s'hauria de localitzar la signatura. Com és lògic pensar, aquest tipus d'àmbit es combina amb el vist en el punt anterior, atesa la dependència que hi ha entre l'estructura interna dels objectes i el seu tipus.

5) Combinacions d'àmbits. Les combinacions d'àmbits són una manera de refinar encara més la porció de l'objecte on podem localitzar la signatura. Per a aconseguir això s'empra de manera simultània algun tipus de combinació dels àmbits descrits anteriorment.

Exemple

Així, una signatura basada en *opcodes* podria especificar que solament es fes la cerca en arxius de tipus executable PE (àmbit de tipus *filtre d'objectes*), en la secció de codi (àmbit de tipus *seccions específiques*), i amb un desplaçament de 1.012 bytes (àmbit de tipus *desplaçament*).

3.2. Detecció semàntica

La detecció semàntica difereix de la detecció sintàctica en fonamentar-se en la identificació d'accions dutes a terme pel programari maliciós, i no per la localització de patrons de bytes en objectes.

Terminologia

La detecció semàntica és també coneguda com a *detecció basada en comportament*.

Identificar aquestes accions i interpretar-ne el significat és una tasca complexa, però que aporta certs avantatges respecte a la detecció sintàctica. En particular, la detecció semàntica és més resistent davant tècniques de mutació com són el polimorfisme o el metamorfisme. Per tant, aquesta estratègia pot ser vista com una manera de superar les deficiències de la detecció sintàctica, atès que una mutació no modifica el comportament final del programari maliciós. Això pot ser entès si considerem també que les signatures sintàctiques no tenen en compte la semàntica de les instruccions.

Vegeu també

El polimorfisme i el metamorfisme s'estudien al subapartat 4.1. d'aquest mòdul.

Les bases de la detecció basada en comportament ja van ser establertes per Cohen (1986, 1987) en els seus primers treballs formals relacionats amb els virus. A pesar que la seva investigació es va centrar en el camp dels virus, aquestes mateixes bases poden ser extrapolades avui dia a qualsevol altre tipus de programari maliciós. En concret, va postular que un virus, igual que qualsevol altre programa, utilitza els serveis proporcionats pel sistema. Per tant, determinar si un programa específic es tracta de programari malintencionat implica establir el que és un ús il·legítim dels serveis del sistema, i contrastar-lo amb les accions fetes per tot procés. D'acord amb l'exposat, Cohen va definir dues aproximacions per a la detecció basada en comportament:

- La primera modelitza el comportament legítim d'acord amb aplicacions conegudes i no malicioses. A partir d'aquí, qualsevol desviació pel que fa a aquesta referència es considera com accions executades per programari maliciós. Aquesta primera aproximació té la capacitat de detectar qualsevol tipus de programari maliciós no conegut. No obstant això, la complexitat d'aquesta proposta és definir aquest model de comportament legítim. Això té sentit especialment si tenim en compte la multitud d'aplicacions que hi ha i les seves diferents naturaleses, la qual cosa condueix al fet que no sigui possible extreure un perfil legítim i únic comú a totes. Per aquest motiu, aquesta estratègia sempre es basa en models estadístics i és procliu a falsos positius.

- La segona aproximació, en contraposició amb l'anterior, es basa a modelitzar el comportament sospitós de les aplicacions malicioses. En aquest cas, qualsevol tipus d'accions detectades que s'aproximin a aquest model de referència induiran a la identificació del codi maliciós. Aquesta segona opció, malgrat no poder detectar nou programari maliciós amb tanta facilitat, s'utilitza de manera més habitual, en no ser tan sensible als falsos positius.

La modelització del comportament en la detecció semàntica implica definir un conjunt de signatures que emprarà el motor de detecció. A diferència de les signatures sintàctiques, les signatures semàntiques requereixen estructures més complexes, en haver de reflectir aspectes dinàmics i de comportament. Aquestes signatures, quan són definides de manera correcta, permeten detectar una àmplia gamma de codi maliciós, inclusivament noves generacions creades a partir d'un programari maliciós particular. Això és possible, ja que programes sintàcticament diferents però amb el mateix comportament poden ser detectats per una única signatura. No obstant això, atès que no és possible una identificació precisa amb les signatures semàntiques, es pot incórrer en un problema en el procés d'eradicació del codi maliciós, especialment quan aquest és de caràcter infecciós com els virus. Addicionalment, un dels avantatges de les signatures semàntiques en comparació de les sintàctiques és el menor nivell de creixement de les seves bases de dades. Recordem que, en el cas de la detecció sintàctica, necessitem signatures addicionals per cada nova forma o variant de programari maliciós, mentre que en la detecció semàntica no és així. Com a conseqüència, la distribució i l'actualització de les bases de dades no és tan freqüent com en la detecció sintàctica.

La recol·lecció d'informació semàntica per a la detecció de programari maliciós es pot fer segons tres contextos diferents:

1) El primer, i el més elemental, implica una captació de la informació en temps real en un sistema. Sota aquesta opció, si a més es registren les accions i els estats intermedis de l'entorn d'execució, és possible retornar el sistema a un estat "saludable" tan aviat com l'amenaça és detectada.

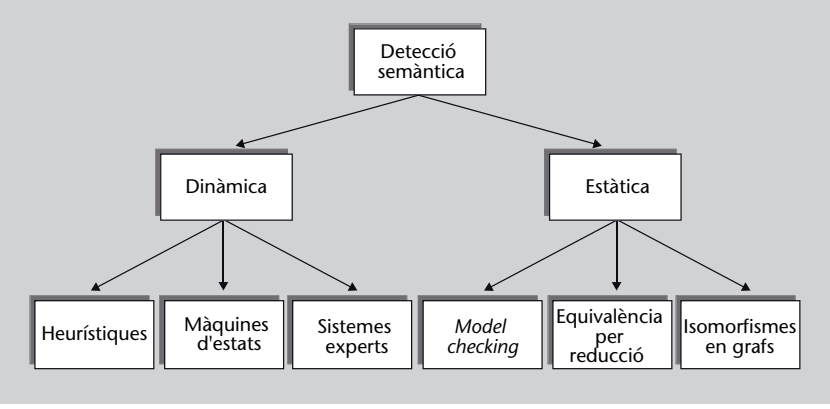
2) Una segona opció implica l'emulació de qualsevol codi abans de l'execució real en un entorn controlat denominat *sandbox*. Les limitacions d'aquesta estratègia apareixen si un codi malintencionat és capaç de detectar el *sandbox*, la qual cosa li permetria adaptar-se i comportar-se com a programari benigne.

3) Finalment, l'ús de màquines virtuals pot ser utilitzat per a la captació de la informació, en ser aquestes capaces de virtualitzar tot un sistema. De nou, si hi ha la possibilitat de detectar la màquina virtual, el programari maliciós en podria modificar el comportament i evadir la detecció.

Lectura recomanada

Hi ha treballs que remarquen el perill de les màquines virtuals i la possibilitat d'infecció de la màquina amfitrió. És el que defensen Embleton i altres (2008); King i altres (2006).

Figura 6. Taxonomia de la detecció semàntica



Independentment del context, la detecció basada en comportament es pot dividir en dues categories principals. La diferència entre aquests dos tipus de detecció és la manera com es capta la informació. D'una banda, l'*anàlisi dinàmica* considera les accions dutes a terme en temps real, mentre que l'*estàtica* obté les accions sense l'execució del codi. Seguidament analitzarem en més profunditat cadascuna d'aquestes dues categories, i veurem quines alternatives hi ha en cadascuna d'acord amb la taxonomia presentada per Jacob, Debar i Filiol (2008).

3.2.1. Anàlisi dinàmica

L'anàlisi dinàmica considera les accions dutes a terme en el sistema per part de tot programa en execució. A partir de la informació d'aquestes accions i una base de coneixement en forma de signatures, el motor de detecció serà capaç de catalogar un procés com a maliciós.

Les accions poden ser analitzades gràcies a la intercepció de les crides al nucli del sistema operatiu, també conegudes com a *syscalls*. Per a això, el motor de detecció s'interposa entre la interfície de crides i el codi de les *syscalls*. D'aquesta manera, cada vegada que un programa fa una crida al nucli, el motor de detecció pren el control abans d'executar el codi de la *syscall* corresponent. Per a la detecció, no solament es té en compte la crida feta, sinó també els seus paràmetres, l'identificador de procés que la va fer, el seu nivell de privilegis, i també qualsevol altra informació de context que pugui ser d'utilitat.

És important remarcar que aquesta mateixa tècnica pot ser utilitzada per programari maliciós que empra tècniques de *rootkits*, la qual cosa podria implicar inhabilitar els mecanismes de detecció. Així mateix, atès que el motor de detecció s'executa abans que les crides al nucli, és de vital importància que la penalització que introdueixi en el sistema sigui mínima. En cas contrari, la percepció que es podria tenir del sistema és el d'una execució lenta.

Vegeu també

Les tècniques *rootkits* s'estudien en el subapartat 4.2..

A partir de la informació de les crides, disposem de diverses alternatives per a tractar aquestes dades i poder detectar el programari malintencionat. En particular, a continuació presentem tres tècniques que difereixen considerablement en els mètodes que empren. En concret, aquestes són: utilització d'**heurístiques**, **màquines d'estats** i **sistemes experts**.

Tècniques heurístiques

Històricament, la utilització d'heurístiques va ser la primera manera de detectar comportaments maliciosos. Aquesta tècnica es basa a interpretar un conjunt d'accions executades en seqüència. Aquestes accions poden ser captura-

des per mitjà de la intercepció de les crides al sistema, usualment mitjançant un *sandbox*. Els motors de detecció heurístics poden seguir dues estratègies:

- La primera estratègia es basa a atorgar a cada acció atòmica un valor quantitatiu en forma de pes. Aquest pes, obtingut a partir de l'experimentació prèvia, expressa la gravetat de l'acció de manera numèrica. En aquest cas, des de la perspectiva del motor de detecció, les signatures es corresponen precisament a l'associació entre accions i pesos. Quan la suma acumulativa d'aquests pesos supera un determinat llindar per a un programa específic, es considera que aquest és codi maliciós.

- La segona tècnica es basa a identificar cada acció atòmica amb una etiqueta en funció d'un coneixement previ. Cadascuna d'aquestes etiquetes estarà associada a un tipus d'acció. El motor de detecció tindrà com a signatures seqüències d'etiquetes que identifiquen diferents tipus de programari maliciós. Quan una seqüència d'accions dutes a terme per un programa origini un conjunt d'etiquetes ordenades i corresponents a una signatura, s'identificarà el codi que les va originar com a maliciós. Per tant, amb cada nova acció serà necessari desar-ne l'etiqueta associada al costat de les anteriors. Usualment, el conjunt de totes les signatures pot ser representat en forma d'arbre, en què cada node es correspon amb una etiqueta, i els nodes fulla identifiquen un programari maliciós particular. D'aquesta manera, la identificació de programari maliciós implica recórrer l'arbre en funció de la seqüència d'etiquetes que es van generant, fins a arribar a un node fulla.

Màquines d'estats

L'ús de màquines d'estats també pot permetre la detecció de programari maliciós sobre la base d'una seqüència d'accions. En particular, els comportaments malintencionats són modelitzats com un autòmat finit determinista $A = (Q,\Sigma,\delta,q_0,F)$ amb les característiques següents:

- Els estats Q de l'autòmat es corresponen amb els estats interns del comportament maliciós.

- L'estat inicial $q_0 \in Q$ es correspon amb l'inici de l'anàlisi.

- L'alfabet Σ està compost per símbols que considerarem crides al sistema.

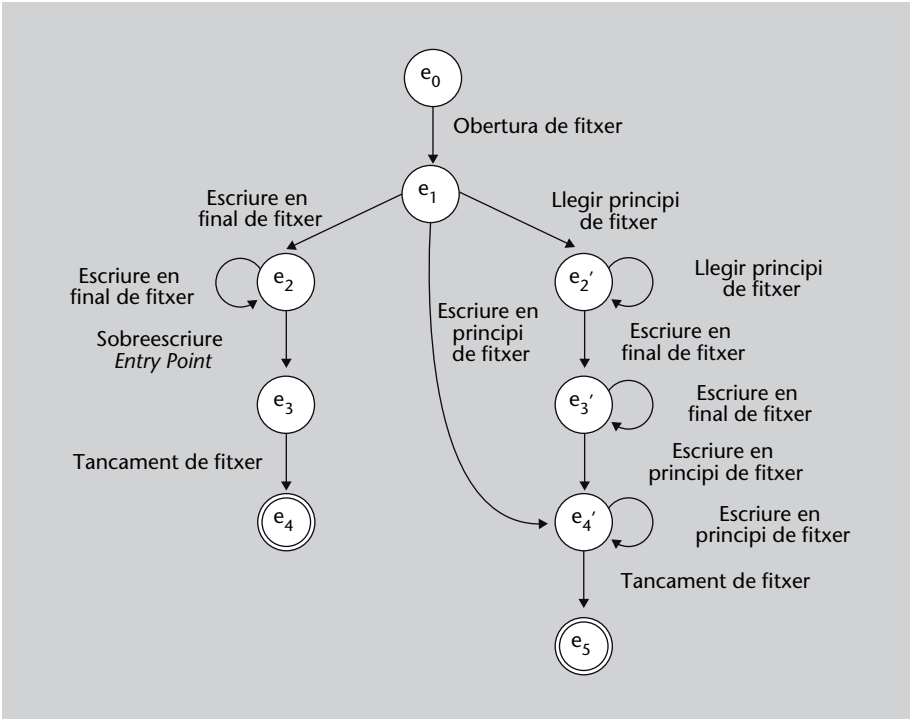
- La funció de transició $\delta : Q \times \Sigma \rightarrow Q$ descriu els canvis d'estat sobre la base de crides sospitoses.

- Els estats de finals $F \subseteq Q$ impliquen la detecció del comportament maliciós que defineix l'autòmat.

A partir d'una instància d'un autòmat determinat que defineix un comportament maliciós i donat un codi en execució, l'autòmat progressarà des de l'estat inicial q_0 sobre la base de les crides que es fan al sistema. Si l'autòmat

acaba en un estat final, el codi associat es considera com a malintencionat. En cas contrari, el codi és catalogat com a benigne. En la figura 7 es pot observar un d'aquests autòmats.

Figura 7. Autòmat finit determinista com a signatura per a la detecció d'un comportament d'infecció vírica (Jacob i altres, 2008)



Sistemes experts

Els sistemes experts es basen en un conjunt de regles modelitzades per un analista per a situacions particulars. En concret, aquestes regles es defineixen per a cada acció sospitosa en termes dels serveis del sistema que s'utilitzen. Associada a cadascuna d'aquestes regles hi ha una decisió sobre l'acció, acceptant-la o denegant-la. Cada vegada que fa una acció de manera individual un procés, aquesta es compara amb el conjunt de regles i, en cas de trobar una regla que concordi, s'executa la decisió vinculada a la regla. En cada acció es té en compte també el nivell de privilegis de qui va llançar l'acció, ja que pot representar la diferència entre una acció legítima d'una altra que no ho és.

3.2.2. Anàlisi estàtica

La detecció basada en comportament utilitzant anàlisi estàtica se sustenta en l'extracció de les accions fetes per un potencial codi maliciós sense l'execució.

A diferència de la dinàmica, l'anàlisi estàtica proporciona més quantitat d'informació i més completa, en no fer l'anàlisi exclusivament sobre accions observades. Per a aconseguir això és necessari extreure la informació semàntica a partir del codi binari, la qual cosa requereix dur a terme diversos processos pera obtenir una representació intermèdia del programa. A partir d'aquesta representació i de les signatures semàntiques es podrà estimar la legitimitat del codi que s'està analitzant.

Per a arribar a aquesta representació intermèdia s'empren tècniques automatitzades d'enginyeria inversa i desassemblament, la qual cosa permet construir tot seguit els grafs de control de flux* i els grafs de flux de dades**. A partir de tots dos grafs es pot obtenir posteriorment un altre tipus de representacions semàntiques d'acord amb el tipus d'anàlisi estàtica que s'estigui fent. Aquests processos no sempre són trivials, atès que el codi maliciós sol incorporar mecanismes que dificulten el procés de desassemblament, com ara tècniques d'ofuscació, antidepuració, etc.

* En anglès, *control flow graph* (CFG).
** En anglès, *data flow graph* (DFG).

Vegeu també

Les tècniques d'evasió s'estudien en l'apartat 4.

Dins de la categoria de l'anàlisi estàtica podem identificar tres tipus de tecnologies diferents per a la detecció del programari maliciós:

1) La verificació de models*. Aquest mètode es basa en una aproximació algebraica, emprant un algorisme que, utilitzant lògica deductiva, simplifica una representació abstracta del codi del programari maliciós usant regles de reescriptura. Per a aconseguir això, inicialment aquesta estratègia ha de fer una transformació del codi a aquesta representació abstracta. És important destacar que les regles de reescriptura preserven la semàntica del codi maliciós, i permeten lluitar contra tècniques de mutació com el polimorfisme i el metamorfisme. Una vegada que s'ha obtingut la forma reduïda, es verifica utilitzant un intèrpret si l'execució exhibeix algun comportament malintencionat d'acord amb especificacions de programari maliciós conegudes, i expressades també en la mateixa representació algebraica.

* En anglès, *model checking*.

2) Equivalència per reducció. Aquesta tècnica se sustenta a utilitzar com a signatures semàntiques fórmules lògiques de primera classe, i que semànticament es corresponen amb accions malicioses. El motor de detecció pren com a entrada el graf de control de flux i les signatures com a fórmules lògiques. Seguidament, verifica quines d'aquestes fórmules corresponen a estats intermedis en tots els possibles camins d'execució del graf. En el cas de detectar la correspondència entre les fórmules i els estats intermedis, l'objecte és catalogat com a maliciós. Atès que hi ha infinits possibles camins d'execució per explorar, l'algorisme és altament recursiu i costós en termes de recursos. De la mateixa manera que la verificació de models, aquesta estratègia permet la detecció independentment de si el programari maliciós utilitza tècniques d'evasió basades en ofuscació de codi.

3) Isomorfismes en grafs. L'isomorfisme en grafs empra el graf de control de flux per a fer la detecció de programari maliciós. En particular, a cada node del graf s'assigna una etiqueta semàntica que descriu el tipus d'acció que exerceix

el bloc bàsic associat. Donada aquesta representació semàntica del potencial codi maliciós, aquesta s'acara amb les signatures semàntiques de la base de dades del motor de detecció. En aquest cas, les signatures semàntiques expressen comportaments maliciosos, i estan representades com un graf, els nodes del qual estan etiquetats amb accions determinades. El procés de detecció implicarà localitzar el graf de la signatura dins de l'extret del codi analitzat. Aquest procés és conegut normalment com el problema de l'isomorfisme en grafs, i consisteix a localitzar un subgraf dins d'un graf major.

4. Mecanismes d'evasió

Sens dubte, el programari especialitzat en la detecció de codi maliciós ha estat una de les vies més emprades contra la lluita del programari maliciós. Si som capaços de detectar un codi maliciós a temps, és possible evitar-ne la infecció. Quan en els inicis els autors de programari maliciós van veure que la proliferació de les seves creacions era contrarestada mitjançant aquest tipus de programari, van haver d'idear noves formes de supervivència. Aquestes estratègies se centren a aconseguir precisament que el programari maliciós romangui ocult als motors de detecció. Com més temps un codi malintencionat passi desapercebut, més temps disposarà per a propagar-se, i la infecció afectarà més sistemes. Addicionalment, aquests mecanismes d'evasió pretenen fer-ne més complexa l'anàlisi, la qual cosa requereix més esforç per part dels experts i, per tant, més temps per a la generació de les signatures.

A causa de l'ampli ús de les estratègies d'evasió per part del programari maliciós, actualment ens veiem en la necessitat d'entendre quins són els mètodes emprats i d'analitzar-los. Per tant, presentem en aquest apartat les tècniques més comunes utilitzades pel programari maliciós amb la finalitat de no ser detectat.

Enllaç d'interès

En la web *VX Heavens* (http://vx.netlux.org) es pot trobar una gran quantitat de documentació sobre programari maliciós i de com és programat. També inclou programari maliciós que pot resultar d'interès per a analitzar-lo i comprendre millor els mecanismes d'evasió.

Podem distingir bàsicament tres tipus de tecnologies d'evasió:

1) les tècniques de *ofuscació*,

2) els mètodes d'*ocultació i autoprotecció* i

3) els mecanismes *antidepuració*.

Malgrat la categorització que exposem aquí, és important remarcar que aquestes no són excloents. És a dir, podem –i de fet, és molt habitual– trobar programari maliciós que combini diverses tècniques de les tres categories de manera simultània. A continuació es descriuen cadascuna de les categories i els diversos mètodes que hi podem trobar.

4.1. Tècniques d'ofuscació

Les tècniques d'ofuscació es poden veure com una transformació del codi d'un programa amb l'objectiu de fer-ne la comprensió més difícil, mentre que al mateix temps se'n preserva la funcionalitat.

Aquestes tècniques han estat aplicades tant a la protecció del programari en general com a la del programari maliciós. Des de la perspectiva de la protecció del programari, l'ofuscació del codi permet protegir els programes d'atacs que atemptin contra la propietat intel·lectual. En concret, aquestes tècniques dificulten aplicar enginyeria inversa, la qual cosa dificulta aquests atacs. D'altra banda, des del punt de vista del programari maliciós, la finalitat dels mecanismes d'ofuscació és doble. En primer lloc, incrementar la dificultat en el procés d'anàlisi i també el temps necessari per a fer-lo i, com a conseqüència, retardar la generació de les signatures. En segon lloc, les estratègies d'ofuscació tenen com a finalitat mutar el codi i, per tant, dificultar o fins i tot fer inviable l'ús de tècniques de detecció basades en signatures sintàctiques.

A continuació presentem les tècniques d'ofuscació més comunes que podem trobar en el programari maliciós, és a dir, el programari maliciós xifrat, l'oligomorfisme, el polimorfisme, el metamorfisme, la compressió d'executables, l'*entry point obscuring* i l'ofuscació per virtualització.

4.1.1. Programari maliciós xifrat, oligomorfisme, polimorfisme i metamorfisme

Aquestes tècniques d'ofuscació es basen a mutar el codi del programari maliciós en cada nova infecció des d'un punt de vista sintàctic, però mantenint la mateixa funcionalitat que la versió prèvia. A partir d'això, es desprèn de manera natural que aquestes tècniques estan centrades sobretot a evadir els motors de detecció sintàctics. No obstant això, com es veurà, aquestes tècniques també dificulten el procés d'anàlisi manual que pugui fer qualsevol expert en seguretat.

Dins d'aquesta categoria trobem diferents possibilitats, en particular el programari maliciós xifrat, l'oligomorfisme, el polimorfisme i el metamorfisme. Cadascuna d'aquestes tècniques és una evolució respecte a l'anterior; el programari maliciós xifrat és la primera a aparèixer cronològicament i, per tant, la més simple, mentre que el metamorfisme es correspon a l'evolució més sofisticada dels quatre mètodes d'ofuscació tractats aquí. Anem a veure'ls amb més detall:

1) Programari maliciós xifrat. Aquest primer sistema evadeix els mètodes de detecció sintàctics mitjançant l'ús d'una funció de xifratge. Concretament, el codi maliciós està compost per una rutina de desxifratge, una clau, i el cos principal del programari maliciós xifrat. Quan s'executa el codi malintencionat, la rutina de desxifratge és qui primer pren el control, desxifrant la resta del cos amb la clau continguda en el codi mateix. Després del desxifratge, el cos principal és executat. En cada nova infecció, el codi maliciós determina una clau aleatòria que usa per a construir una nova variant. Aquesta nova versió estarà composta per la mateixa rutina de desxifratge, la clau obtinguda aleatòriament, i el cos del programari maliciós xifrat amb la nova clau. D'aquesta

Instruccions per al xifratge/desxifratge

El programari maliciós que empra rutines de desxifratge com a forma d'ofuscació pot utilitzar diversos mètodes, com són l'ús de les instruccions invertibles `ADD`, `SUB`, `INC`, `DEC`, `XOR` o `NOT`, o combinacions d'aquestes.

manera, el cos del codi maliciós és diferent cada vegada, fet que impedeix determinar signatures sintàctiques basades en aquest cos. No obstant això, atès que la rutina de desxifratge roman constant al llarg de totes les generacions, és possible determinar signatures sintàctiques associades a la porció de codi responsable del desxifratge.

2) Oligomorfisme. Amb l'objectiu de superar les deficiències del programari maliciós xifrat, els creadors de programari maliciós van idear una evolució d'aquest que denominem oligomorfisme. A diferència del programari maliciós xifrat, en el qual la funció de desxifratge roman constant, l'oligomorfisme es basa a modificar la rutina de desxifratge generació rere generació. Per a això, el codi malintencionat crea dinàmicament una nova rutina basada en porcions de codi seleccionades aleatòriament entre un conjunt d'alternatives. Com és lògic pensar, el xifratge del cos es fa segons aquesta rutina construïda. En aquest sentit, l'oligomorfisme es considera el precursor del que es coneixeria posteriorment com a polimorfisme. Malgrat l'evolució que representa aquesta idea respecte al programari maliciós xifrat, el nombre de possibilitats quant a rutines de desxifratge és limitat i, per tant, utilitzar una detecció mitjançant signatures sintàctiques sobre el codi de desxifratge encara continua essent una estratègia factible. Així, una possibilitat és crear signatures sintàctiques per a cada porció de codi entre les diverses alternatives i, posteriorment, intentar detectar diverses d'aquestes signatures encadenades, que compondran la rutina de desxifratge. Una altra alternativa per a la detecció de programari maliciós que incorpora aquest mecanisme és l'emulació del codi. Això permet el desxifratge dinàmic del cos del programari maliciós i, *a posteriori*, localitzar una signatura sintàctica definida.

3) Polimorfisme. Com a millora a l'oligomorfisme des de la perspectiva de l'evasió, el programari maliciós polimòrfic va aparèixer posteriorment. Aquest incorpora la capacitat de generar una gran quantitat –vora milions– de rutines de desxifratge diferents. Per a aconseguir això, el programari maliciós polimòrfic es recolza en l'ús de diversos mètodes d'ofuscació, com ara la reordenació de rutines o la inserció de codi innecessari. Aquests mètodes, comuns també al metamorfisme, seran tractats més endavant. Com a conseqüència de la gran quantitat de rutines de desxifratge que un programari maliciós pot crear en cada nova generació, no hi ha un patró sintàctic per buscar. La detecció d'aquest tipus de codi maliciós requereix mecanismes més sofisticats, com són l'ús d'emuladors, o la detecció basada en comportament.

Lectura complementària

L'estratègia del xifratge/desxifratge com a mecanisme d'ofuscació pot ser usada de diverses maneres sofisticades. Així, per exemple, és possible trobar programari maliciós amb diversos nivells de xifratge niats amb les rutines de desxifratge respectives. Us animem a llegir Szor (2005) per a ampliar aquesta informació.

4) Metamorfisme. El programari maliciós metamòrfic va aparèixer com un mètode d'ofuscació que anava més enllà de les propostes anteriors quant a nivell de sofisticació. Igual que el polimorfisme, el metamorfisme empra diverses tècniques d'ofuscació; no obstant això, en el cas que ens ocupa s'apliquen a la totalitat del cos del programari maliciós, no a una rutina de desxifratge. D'aquesta manera, cada nova versió creada per a una infecció condueix a una variant totalment diferent a l'anterior sintàcticament parlant. Això requereix que el programari maliciós sigui capaç de reconèixer el seu propi cos principal,

analitzar-ho i mutar-lo quan es propagui. En certa manera, podem dir que el programari maliciós metamòrfic és pseudoconscient de si mateix.

Tècniques d'ofuscació del polimorfisme i del metamorfisme

Com hem comentat anteriorment, tant el polimorfisme com el metamorfisme empren tècniques d'ofuscació específiques. En el cas del polimorfisme, aquestes estratègies s'usen per a mutar la rutina de desxifratge, mentre que en el metamorfisme s'usen per a mutar la totalitat del cos del codi malintencionat. En tots dos casos, les tècniques se solen combinar de manera simultània, i s'aconsegueix així que la detecció sintàctica sigui més difícil. Amb la finalitat de comprendre millor com actuen aquests tipus de codis maliciosos en el procés de crear noves generacions mutades, anem a introduir les tècniques d'ofuscació principals que utilitzen:

1) Inserció de codi innecessari. Aquesta tècnica insereix instruccions (codi brossa) en el cos principal del programari maliciós que no tenen cap tipus d'efecte en el comportament final del codi. Això permet canviar l'aparença sintàctica del programari maliciós alhora que en manté la funcionalitat.

2) Reassignació de registres. Aquest mètode d'ofuscació es basa que el mateix programari maliciós analitza el seu cos principal i crea una nova versió en la qual es reassignen nous registres a les instruccions que els empren. Aquest procés es fa respectant la lògica del programa i mantenint la consistència en relació amb l'ús dels registres per part de les instruccions. D'aquesta manera, els *opcodes* de la nova versió resultant difereixen respecte a l'anterior.

3) Substitució de codi per instruccions equivalents. Aquesta estratègia es fonamenta a crear una nova generació reemplaçant un conjunt d'instruccions de la versió anterior per altres totalment equivalents. Així, mentre que la lògica del programa es manté, el seu aspecte es veu modificat en comparació de la versió prèvia.

4) Permutació de subrutines. Aquesta tàctica ofusca el codi original canviant l'ordre en el qual apareixen les subrutines dins del cos del programari maliciós. De manera numèrica, si un programari maliciós està compost per n subrutines diferents, aquest serà capaç de generar un total de $n!$ variants.

5) Transposició de codi. Aquest mecanisme muta el codi en traslladar blocs d'instruccions mentre es preserva el comportament original del programari maliciós. Hi ha dues maneres d'aconseguir-ho. La primera es basa a barrejar els blocs d'instruccions de manera aleatòria i introduir salts incondicionals al final de cada bloc per mantenir l'ordre d'execució. La segona forma aconsegueix mutar el codi en seleccionar i traslladar instruccions independents l'ordre d'execució de les quals no afecta el comportament final.

6) Integració de codi entrellaçat. Aquesta forma d'evasió és emprada per programari maliciós de propagació per infecció vírica, el qual s'insereix a si

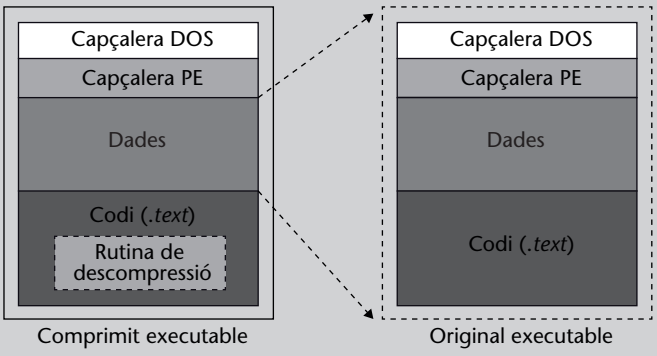
mateix en el codi de l'arxiu per infectar de manera entrellaçada. Per a això, és necessari que el programari maliciós desassembli l'executable per infectar i el representi en un format manejable, s'insereixi entre el codi de l'executable, modifiqui el codi original per mantenir la consistència a escala d'instruccions i de referència a dades i, finalment, reconstrueixi el nou executable infectat.

4.1.2. Compressió d'executables

L'origen de la compressió dels executables es remunta als anys vuitanta, quan la capacitat dels sistemes d'emmagatzematge d'informació era menor que avui dia i el cost més elevat. Aleshores, i amb l'objectiu d'obtenir un aprofitament millor de l'espai, es van utilitzar algorismes de compressió sobre els executables.

Si bé és cert que aquesta idea pot continuar essent aplicada actualment de manera legítima, aquesta estratègia també pot ser utilitzada per programari maliciós com a mecanisme d'evasió. El motiu d'això es troba en la idea següent. Si un codi maliciós es modifica lleugerament i s'allibera com una nova versió, és probable que comparteixi porcions de codi respecte al seu predecessor i que, per tant, l'existència d'una signatura sintàctica que detectava la generació prèvia probablement també detecti la nova. No obstant això, si s'utilitza un algorisme de compressió sobre la nova versió, encara que aquesta incorpori un simple canvi, resultarà en un executable radicalment diferent. D'aquesta manera, la signatura sintàctica que detectava la versió anterior no és útil per a variants futures.

Figura 8. Representació conceptual d'un programari maliciós empaquetat en Windows



Un executable comprimit està estructurat en dues grans parts. D'una banda, la rutina de descompressió i, de l'altra, les dades comprimides que es corresponen amb el codi original de l'executable. Quan es llança l'execució d'un programa comprimit, la rutina responsable de la descompressió s'executa en primera instància. Llavors, aquesta, a partir de les dades comprimides, regenerarà el codi de l'executable original i li cedirà el control.

A pesar que cada creador de programari maliciós podria implementar els seus propis algorismes de compressió, la realitat mostra que actualment hi ha múltiples eines que automatitzen aquest procés, i que són emprades assíduament pels creadors de programari maliciós. Aquestes són conegudes normalment amb el nom d'empaquetadors (o *packers* en anglès). La seva manera de treballar és la següent: donat un executable indicat per l'usuari, generen una nova versió composta pel codi original comprimit, i també la rutina necessària per a la descompressió. Algunes d'aquestes eines fins i tot afegeixen tècniques antidepuració a l'executable resultant, la qual cosa en dificulta l'anàlisi posterior.

Packers

A Internet podeu localitzar informació sobre els *packers* més coneguts. Alguns d'aquests són UPX, Armadillo, ASPack, PE Compact o ASProtect, entre altres.

4.1.3. *Entry point obscuring*

L'*entry point obscuring* (EPO) és un mètode d'ofuscació propi del programari maliciós d'infecció. Tradicionalment, la infecció d'un executable sempre s'ha fet modificant-ne el punt inicial d'execució (o *entry point*), de manera que aquest apunti abans al codi del programari maliciós. Una vegada que el codi maliciós pren el control i fa les accions que l'interessen, cedeix el control al codi de l'executable original. A diferència d'això, la tècnica d'*entry point obscuring* modifica l'executable original en qualsevol punt del seu codi, inserint instruccions de tipus `CALL` o `JMP` per a redirigir el flux d'execució cap al codi del programari maliciós. Fent això, s'aconsegueix evadir motors de detecció sintàctics que intentin localitzar programari maliciós analitzant l'*entry point* com a àmbit de cerca.

4.1.4. Ofuscació per virtualització

L'ofuscació per virtualització és un mecanisme d'ofuscació que implementa en el codi mateix del programari maliciós un entorn d'execució al costat d'un intèrpret, el qual és capaç d'executar programes escrits en un llenguatge específic en forma de *bytecodes*. L'intèrpret –altament ofuscat– accepta un llenguatge que es tria aleatòriament en cada infecció. Dit d'una altra manera, podem considerar que el programari maliciós que inclou aquesta estratègia implementa un processador virtual que accepta un repertori d'instruccions particular.

Terminologia

En la bibliografia, en referència a l'estratègia d'ofuscació per virtualització també trobarem la denominació *ofuscació mitjançant màquines virtuals* (Rolles, 2009).

La manera d'operar d'aquest tipus de programari maliciós es basa a seleccionar aleatòriament, abans d'una nova infecció, un llenguatge acceptat per l'intèrpret, i recodificar el cos del programari maliciós sobre la base d'aquest nou llenguatge. Evidentment, l'intèrpret que contindrà la nova variant es genera d'acord amb el nou llenguatge triat. D'aquesta manera, tant l'intèrpret com el conjunt de *bytecodes* es modificaran sintàcticament, mentre que es preserva el comportament del codi maliciós.

Fer una anàlisi d'aquest tipus de codi maliciós per a entendre'n el funcionament implica comprendre l'arquitectura i el processador virtual que imple-

menta. Això no sempre és trivial i es pot arribar a convertir en un treball tediós per a l'analista. L'ús d'enginyeria inversa en aquesta situació es converteix en una cosa complexa, en haver d'abordar aspectes d'alt nivell (per exemple, l'arquitectura i el processador virtual mateixos), que queden ofuscats pels detalls de baix nivell. Fins i tot després de l'esforç per a comprendre el llenguatge i l'intèrpret, una nova versió pot aparèixer amb una implementació diferent d'un processador virtual nou.

4.2. Tècniques d'ocultació i autoprotecció

Les tècniques d'ocultació i autoprotecció són mecanismes que implementa el programari maliciós amb l'objectiu de passar desapercebut tant a administradors com a programari de detecció, o bé per a protegir-se i dificultar-ne l'eradicació en un sistema infectat. Normalment, el conjunt de tècniques que permeten aconseguir això es denominen *mecanismes rootkit*.

Lectura recomanada

Una bona referència per a entendre com funcionen les tècniques de *rootkit* en les plataformes Windows és el llibre següent:
G. Hoglund; J. Butler (2005). *Rootkits: Subverting the Windows Kernel*. Addison-Wesley

Per a aconseguir ocultar-se o protegir-se, el codi maliciós modifica parts del sistema operatiu. A pesar que cada sistema operatiu té les seves particularitats i aquestes modificacions es poden fer de diverses maneres, de manera general podem considerar que hi ha tres maneres d'aconseguir l'ocultació o autoprotecció. Aquestes tres maneres són les següents:

1) Mecanismes *rootkit* en espai d'usuari. Els mecanismes *rootkit* a escala d'aplicació ataquen les API* utilitzades pels programes. Aquestes API estan implementades com a biblioteques dinàmiques proporcionades pel sistema operatiu mateix, la qual cosa permet compartir-ne el codi entre diverses aplicacions de manera simultània. D'aquesta manera, les aplicacions tenen accés a un conjunt de funcions que no han d'implementar, ja que estan proporcionades pel sistema mateix. Evidentment, aquest principi parteix de la premissa que les aplicacions confien plenament en les funcions de les biblioteques. En el context que ens ocupa, això té especial sentit si tenim en compte que algunes d'aquestes funcions proporcionen mètodes per a l'obertura d'arxius, o l'obtenció de la llista de processos del sistema. Així, si un programari maliciós és capaç d'interceptar la crida d'una funció per a obtenir la llista de processos actius, per exemple, es podria assegurar que el resultat proporcionat no incorporés el procés associat a ell mateix, la qual cosa li permetria passar desapercebut a les aplicacions que utilitzessin aquesta funció.

*** API és la sigla d'*application program interface*.**

Per a entendre com el programari maliciós pot aconseguir això, hem de comprendre de quina manera funcionen les biblioteques dinàmiques. Quan es genera un executable d'un programa que utilitza biblioteques dinàmiques, el compilador inclou en la imatge binària resultant certes estructures de dades. Aquestes estructures de dades incorporen el nom de les biblioteques al costat de la llista de funcions que usa el programa. Així mateix, per a cada element de la llista hi ha un apuntador no inicialitzat amb l'adreça de memòria de la funció associada. Quan un programa es carrega en memòria per

a executar-lo, el sistema operatiu llegeix les estructures de la imatge binària. Seguidament, amb la informació pertinent de les estructures, carrega les biblioteques en memòria, estima les adreces de cada funció per a cada biblioteca, i emplena els apuntadors de la llista amb les adreces de les funcions. D'aquesta manera, quan en el procés corresponent es fa una crida a una funció d'una biblioteca, se sap l'adreça de memòria on aquesta es troba i, per tant, cap a on s'ha de redirigir el flux d'execució. Amb la modificació d'un d'aquests apuntadors, el programari maliciós pot desviar el flux d'execució al seu codi propi, la qual cosa li permet tenir ple control sobre la crida i falsejar les dades de resposta. La modificació de l'apuntador d'una funció d'una biblioteca es pot fer de diverses maneres, i les tècniques depenen ja de cada sistema operatiu.

2) Mecanismes *rootkit* en espai de nucli. Conceptualment, els mecanismes de *rootkits* en espai de nucli no s'allunyen substancialment dels d'espai d'usuari. En aquest sentit, aquest tipus de mecanismes també desvia l'execució cap a codi maliciós del seu interès, el qual actua en funció de les seves necessitats. La diferència principal entre les dues categories es basa que en aquesta segona el procés de desviar el flux d'execució es fa amb més nivell de privilegis. Concretament, el codi malintencionat es carrega i s'executa en l'espai d'adreces del nucli del sistema operatiu. Precisament, aquesta característica d'execució amb privilegis de sistema els proporciona més resistència a ser detectats o eliminats, ja que les eines de detecció o eliminació se solen executar en espai d'usuari. Això implica més complexitat en el seu codi, la qual cosa els fa menys comuns.

La manera més habitual de desviar el flux d'execució per part d'aquest tipus de mecanismes es basa a modificar les taules que contenen els apuntadors a les crides del nucli (*syscall table*). Amb el canvi dels apuntadors de les *syscalls*, el codi maliciós tindrà ple control sobre les crides que poden representar un risc per a ell. D'aquesta manera, el programari maliciós actuarà en conseqüència cada vegada que des de l'espai d'usuari es faci una crida interceptada.

Exemple

Així, si s'intenta executar la crida al nucli per a matar el procés del programari maliciós, i aquesta crida està interceptada, el codi maliciós pot actuar perquè això no succeeixi. No obstant això, si la crida per a matar un procés es fa sobre un de diferent, el codi malintencionat cridarà la *syscall* original. D'aquesta manera, el sistema es comportarà de manera normal per als casos que no atemptin contra el programari maliciós.

Malgrat que la tècnica de la modificació de la taula de les *syscalls* és la més comuna, hi ha altres alternatives, com és l'ús de controladors per a l'ocultació d'informació o l'autoprotecció.

3) Mecanismes *rootkit* híbrids. Els mecanismes híbrids, tal com indica el seu nom, poden ser aplicats tant en espai d'usuari com en espai de nucli. Aquesta tècnica, a diferència de les dues anteriors, en les quals es reemplaça algun tipus d'apuntador, ataca directament la funció implicada. Per a això, modifica els primers bytes de la funció, reemplaçant-los per un salt incondicional cap al codi del programari maliciós. D'aquesta manera, es modifica el flux d'execució i pren el control el programari maliciós quan es produeix una crida a

una funció. Per a preservar el comportament original de la crida, i abans de la sobreescriptura dels primers bytes amb el salt incondicional, el codi malintencionat ha de desar els bytes reemplaçats. Aquest mecanisme resulta encara més complex de detectar si el programari maliciós introdueix el salt incondicional en qualsevol altre punt de la funció diferent dels seus primers bytes.

4.3. Mecanismes antidepuració

Els mecanismes antidepuració són un conjunt d'estratègies que el programari maliciós pot incorporar en el seu propi codi, la missió de les quals és la de dificultar qualsevol procés d'enginyeria inversa que s'intenti aplicar sobre aquest. El seu objectiu principal és el de detectar si un depurador està supervisant l'execució del codi maliciós mateix. Si aquest és el cas, el programari maliciós mateix modificarà el seu comportament.

Lectura recomanada

Una bona recopilació de tècniques antidepuració per a la plataforma Windows es pot trobar en l'article "Anti-Debugging. A Developers View", de Tyler Shields, que està disponible en aquesta adreça d'Internet: http://www.veracode.com/images/pdf/whitepaper_antidebugging.pdf

En aquest sentit, pot adoptar diverses postures:

- executar codi complex sense cap finalitat per descoratjar l'analista,
- exhibir un comportament legítim en comptes de maliciós, o
- finalitzar l'execució.

Les tècniques més recents fins i tot empren algun mètode per a detectar si l'execució s'està fent en una màquina virtual, atès que és molt habitual usarne en els processos d'anàlisis de programari malintencionat. Cal destacar que la majoria dels mecanismes emprats són altament dependents dels sistemes operatius i de les arquitectures dels sistemes.

Resum

En els últims anys, la indústria del programari maliciós ha focalitzat els seus esforços a explotar les deficiències de seguretat com una possible via d'infecció dels sistemes. Entre totes les possibles deficiències, els desbordaments de memòria intermèdia són una clara amenaça, en permetre als atacants l'execució de codi arbitrari. La causa d'això es troba en els errors comesos pels programadors en no controlar els límits de les mides de les memòries intermèdies, la qual cosa permet sobreescriure les adreces de retorn emmagatzemades en la pila, i redirigir el flux d'execució cap a una seqüència d'instruccions especialment preparada.

En l'àmbit del programari maliciós, establir una taxonomia genèrica que permeti classificar el codi maliciós sobre la base de certes característiques es fa complex. Cada dia es fa més palès que les noves formes de programari maliciós exhibeixen comportaments que donen lloc a classificar-se segons diverses categories. Això posa de manifest la constant evolució que està sofrint el codi maliciós, i la necessitat d'haver d'extremar les mesures de seguretat per a protegir als sistemes d'informació. En aquest sentit, el programari de detecció de programari maliciós és una eina proactiva que ha de ser combinada al costat d'altres estratègies, com són les bones pràctiques dutes a terme pels usuaris, els dissenys de programari basats en patrons de protecció, o l'ús de les signatures digitals.

El programari de detecció de codi maliciós se sustenta en dos grans pilars per a la localització d'objectes nocius. La forma més bàsica és la detecció basada en la sintaxi del codi; aquesta estratègia és un mètode fàcilment eludible pel codi maliciós per mitjà de tècniques d'ofuscació. D'altra banda, la detecció semàntica es presenta com una manera de superar les deficiències de la detecció sintàctica, i fins i tot és capaç de detectar programari maliciós desconegut. Malgrat la indiscutible utilitat del programari de detecció, és important remarcar que la detecció perfecta no existeix, tal com ja va postular Cohen en els seus treballs en el camp dels virus informàtics. Com a conseqüència, els nostres sistemes sempre han de ser considerats com a potencialment vulnerables a qualsevol forma de codi maliciós.

Una prova que la detecció perfecta no existeix és l'existència d'una gran quantitat d'estratègies implementades pel programari maliciós. Així, les tècniques d'ofuscació, els mecanismes *rootkit*, o els mètodes antidepuració, són un clar exemple d'aquestes tàctiques. Sens dubte, totes aquestes tècniques estan pensades per a eludir els motors de detecció, o com a mesura de protecció per a evitar-ne l'eradicació. D'aquesta manera, el codi maliciós és capaç de perdurar

sigilosament en els sistemes. Aquest fet hauria de fer-nos reflexionar sobre la importància que té el programari maliciós com a amenaça, sempre latent, en el camp de la seguretat informàtica.

Activitats

1. Descarregueu el programari de compressió per a executables UPX (*Ultimate Packer for eXecutables*) que trobareu en aquesta adreça web: http://upx.sf.net/. Copieu un executable qualsevol en un directori i comprimiu-lo amb aquest programari. Observeu la diferència de mida. Utilitzeu un editor hexadecimal (per exemple, l'HT Editor*) per a veure les diferències entre els dos executables de nivell de contingut binari, i de les capçaleres PE.

* http://hte.sf.net/

2. Analitzeu algun antivirus i busqueu si implementa algun tipus d'heurística Està activada per defecte? Quin tipus de relació hi ha entre les heurístiques i els falsos positius?

3. Llegiu el document *Creating signatures for ClamAV*, que podreu localitzar en aquesta adreça web: http://www.clamav.net/doc/latest/signatures.pdf. S'hi descriu la manera d'especificar signatures sintàctiques per a la detecció de programari maliciós en l'antivirus *ClamAV*. Quin tipus de signatures permet? Té la possibilitat d'indicar àmbits de cerca associats a les signatures? Quins tipus d'àmbits suporta?

4. Busqueu a Internet la descripció d'algun programari maliciós específic que hagi explotat algun desbordament de memòria intermèdia. Classifiqueu el programari maliciós localitzat segons la taxonomia que hem presentat.

5. Raoneu una llista de bones pràctiques que tot usuari hauria de seguir per a reduir el risc d'infecció dels sistemes amb programari maliciós. Creieu que la gent del vostre entorn segueix aquestes bones pràctiques totalment?

6. Localitzeu en el portal http://www.exploit-db.com/ diversos *shellcodes* i identifiqueu quina acció permeten fer i per a quina plataforma són vàlids. Verifiqueu que cap no conté un *opcode* amb el valor `0x00`.

7. En la mateixa web de l'activitat anterior, busqueu un *exploit* que permeti l'execució remota de codi. Identifiqueu quin és el programari que es veu afectat per l'*exploit*, i intenteu comprendre on es produeix el desbordament de memòria intermèdia.

Exercicis d'autoavaluació

1. L'execució de codi arbitrari en una explotació d'un desbordament de memòria intermèdia es produeix en la fase de...
a) pròleg.
b) crida.
c) retorn.
d) Totes les anteriors.

2. Un *shellcode* utilitzat en un desbordament d'una memòria intermèdia...
a) sol estar precedit d'instruccions `NOP` per a maximitzar l'èxit de l'atac.
b) té una mida limitada quan es tracta d'un *stack overflow*.
c) té una forta dependència de l'arquitectura i del sistema operatiu de la plataforma atacada.
d) Totes les anteriors.

3. L'execució de codi gràcies a un desbordament d'una memòria intermèdia és possible, ja que...
a) se sobreescriu l'adreça de retorn d'alguna funció.
b) hi ha llenguatges de programació que operen amb memòries intermèdies sense controlar-ne les seves grandàries.
c) es poden crear *shellcodes* que no continguin cap *opcode* amb valor `0x00`.
d) Totes les anteriors.

4. Quin de les afirmacions següents és falsa respecte a la detecció de programari maliciós amb signatures sintàctiques?
a) Les signatures poden incloure expressions regulars.
b) Les signatures poden ser definides sobre la base de funcions resum criptogràfiques.
c) Les signatures són completament immunes al polimorfisme.
d) Cap de les anteriors.

5. La detecció perfecta de programari maliciós...
a) és un problema indecidible.
b) sempre és possible amb un motor de detecció basat en anàlisi semàntica.
c) sempre és possible amb un motor de detecció basat en signatures sintàctiques.
d) Cap de les anteriors.

6. L'empaquetament emprat pel programari maliciós...
a) és una tècnica antidepuració.
b) és una forma d'evasió contra la detecció semàntica.
c) Les signatures sintàctiques són immunes al programari maliciós empaquetat.
d) Cap de les anteriors.

7. Quina afirmació és certa respecte al programari maliciós de propagació com a cuc?
a) Pot utilitzar l'enginyeria social en un correu per a propagar-se.
b) Pot utilitzar un desbordament de memòria intermèdia per a propagar-se.
c) Pot implementar tècniques *rootkit*.
d) Totes les anteriors.

8. Els àmbits de cerca utilitzats en les signatures sintàctiques...
a) acceleren el procés de detecció en termes generals.
b) penalitzen en rendiment en la identificació de codi maliciós.
c) de vegades poden ser evadits mitjançant *entry point obscuring*.
d) *a* i *c*

9. La reassignació de registres és una tècnica d'ofuscació pròpia...
a) del programari maliciós xifrat.
b) del programari maliciós oligomòrfic.
c) del programari maliciós polimòrfic.
d) Cap de les anteriors.

10. Quina afirmació és certa en relació amb la detecció basada en comportament?
a) No requereix signatures.
b) La base de dades té un nivell de creixement inferior en comparació de la de les signatures sintàctiques.
c) Són molt eficients en l'eradicació del programari maliciós, atès el seu nivell d'identificació precisa.
d) Cap de les anteriors.

Solucionari

Exercicis d'autoavaluació

1. c; **2.** d; **3.** d; **4.** c; **5.** a; **6.** d; **7.** d; **8.** d; **9.** c; **10.** b;

Bibliografia

Cohen, F. B. (1984). *Computer viruses: Theory and experiments*. Los Angeles (EE. UU.): University of Southern California.

Cohen, F. B. (1986). *Computer viruses*. Tesi Doctoral, University of Southern California, Los Angeles (EE. UU.).

Cohen, F. B. (1987). «Computer viruses: Theory and experiments». *Computers & Security*, volum 6 (núm. 1, pàgs. 22–35).

Embleton, S.; Sparks, S.; Zou, C. (2008). «SMM Rootkits: a New Nreed of OS Independent Malware». A: «Proceedings of the 4th international conference on Security and privacy in communication netowrks», SecureComm '08, (pàgs. 11:1–11:12). New York (EE. UU.): ACM.

Filiol, E. (2007). «Metamorphism, Formal Grammars and Undecidable Code Mutation». *International Journal of Computer Science*, (núm. 2, pàgs. 70–75).

Foster, J. C.; Osipov, V.; Bhalla, N. (2005). *Buffer Overflow Attacks: Detect, Exploit, Prevent*. (1a. ed.). Syngress.

Jacob, G.; Debar, H.; Filiol, E. (2008). «Behavioral detection of malware: from a survey towards an established taxonomy». *Journal in Computer Virology*, (núm. 4, pàgs. 251–266).

King, S. T.; Chen, P. M.; Wang, Y. M.; Verbowski, C.; Wang, H. J.; Lorch, J. R. (2006). «SubVirt: Implementing malware with virtual machines». A: «SP'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy», (pàgs. 314–327). Washington (EE. UU): IEEE Computer Society.

Koziol, J.; Litchfield, D.; Aitel, D.; Anley, C.; Eren, S.; Mehta, N.; Hassell, R. (2004). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley.

Kramer, S.; Bradfield, J. (2010). «A General Definition of Malware». *Journal in Computer Virology*, (núm. 6, pàgs. 105–114). Disponible online: http://dx.doi.org/10.1007/s11416-009-0137-1.

Rolles, R. (2009). «Unpacking Virtualization Obfuscators». A: «Proceedings of the 3rd USENIX conference on Offensive technologies», WOOT'09. Berkeley (EE. UU.): USENIX Association.

Spinellis, D. (2003). «Reliable Identification of Bounded-length Viruses is NP-complete». *IEEE Transactions on Information Theory*, volum 49 (núm. 1, pàgs. 280–284).

Szor, P. (2005). *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.