Principis de Ciberseguretat

Octubre 2022

Índex

1 Introducció

Avui en dia escoltem el terme ciberseguretat de manera bastant habitual en els mitjans de comunicació ja que constantment apareixen notícies sobre fugides de dades, atacs informàtics, regulacions de privacitat, virus massius, etc. i és que la tecnologia té cada vegada més rellevància a les nostres vides. Per això, en aquesta pràctica farem una breu aproximació al concepte de ciberseguretat, com s'aborda i uns passos inicials per a començar entendre com es fa un ciberatac.

Els programes compilats a fitxers executables binaris poden contenir errors o forats que poden desencadenar un comportament no desitjat pels seus autors. En entendre acuradament l'entorn on s'executen els programes, les instruccions i la memòria, un atacant pot elaborar amb gràcia una entrada específica, adaptada per desencadenar aquests comportaments i un control no desitjat sobre la lògica original del programa. Una de les maneres d'aconseguir-ho és corrompent els valors crítics de la memòria. Aquest pràctica se centra en algunes tècniques que es poden utilitzar per aprofitar els *buffer overflows* així com altres vulnerabilitats de corrupció de memòria per treure profit dels binaris compilats.

La pràctica que es presenta aquí es basa en el Treball de Fi de Grau d'Oriol Ornaque Blázquez titulat "Binary exploitation" que va ser presentat a la primavera del 2021 (`http://hdl.handle.net/2445/179863`).

Nota: aquesta pràctica ha sigut testejada satisfactòriament a la màquina virtual de la asignatura. Hauria de funcionar també a altres sistemes Linux. A les aules no hi ha instal·lat, per motius de seguretat, el programari que es fa servir a la secció 2.2.

```c
void funcio()
{
  int a[1000];
  int b;

  b = 5;

  printf("Direccio de b: %p\n", &b);
  printf("Valor de b: %p\n", b);

  printf("Direccio de a[1]: %p\n", &a[1]);
  printf("Direccio de a[0]: %p\n", &a[0]);
  printf("Direccio de a[-1]: %p\n", &a[-1]);

  printf("Modifico el valor: a[-1] = 10\n");
  a[-1] = 10;

  printf("Valor de b: %d\n", b);
}

int main(void)
{
  funcio();
}
```

Figura 1: Modificació de variables en cas que hi hagi un error de programació.

2 Stack overflow

La forma més comuna per a les CPU d'implementar crides a procediments o subrutines és per mitjà d'una pila. La pila és una solució senzilla i efectiva per fer un seguiment de l'ordre en que es fan les crides: quan es crida a una subrutina, s'emmagatzema a la pila la direcció de memòria de la següent instrucció a executar en retornar la crida a la subrutina. Quan la subrutina hagi acabat d'executar fa servir la informació que hi ha emmagatzemada a la pila per continuar l'execució de l'aplicació. Aquesta la direcció que s'emmagatzema a la pila s'anomena la direcció *de retorn.*

2.1 Sobreescriptura de la direcció de retorn

Pot modificar un atacant la direcció de retorn de la pila? Sí, ho pot fer!

Abans de fer-ho vegem un exemple que mostra com es poden modificar els valors de variables de la pila en cas que hi hagi un error de programació, vegeu la figura 1 que correspon al codi `pila_modificacio_variable.c`. Compileu el codi sense opcions d'optimització, i.e. `-O0`, i executeu el codi. Observeu que en modificar el valor d'una variable es poden modificar valors d'altres variables.

Pregunta Com és que s'ha pogut modificar el valor de la variable b sobreescrivint el valor d'a[-1]? Com s'emmagatzemen les variables a la pila perquè això pugui succeir? Feu-vos un dibuix perquè us quedi clar ja que ara en traurem profit!

L'objectiu és aprofitar l'estructura de la pila per tal de sobreescriure la direcció de retorn de la pila. A la figura es mostra un exemple que s'ha obtingut de https://exploit.education/

```c
#define LEVELNAME "phoenix/stack-four"

#define BANNER \
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

char *gets(char *);

void complete_level() {
  printf("Congratulations, you've finished " LEVELNAME " :-) Well done!\n");
  exit(0);
}

void start_level() {
  char buffer[64];
  void *ret;

  gets(buffer);

  ret = __builtin_return_address(0);
  printf("and will be returning to %p\n", ret);
}

int main(int argc, char **argv) {
  printf("%s\n", BANNER);
  start_level(argv[1], argv[2]);
}
```

Figura 2: Codi `stack4.c` de exploit.education.

anomenat `stack4`, vegeu figura 2. L'objectiu en aquest exemple es aconseguir executar la funció *complete_ level* que no es crida en cap moment!

Observeu que hi ha un buffer de 64 bytes a la pila i que es permet a l'usuari introduir una cadena fent servir la funció *gets*. Si mireu el manual d'aquesta funció veureu que hi diu "Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead."

És a dir, que en aquest exemple la funció *gets* no comprovarà si s'introdueixen més de 64 bytes. Si un usuari introdueix més de 64 bytes, produirà un *stack overflow*. És a dir, escriurà fora del vector podent sobreescriure la direcció de retorn de la funció. Aquest "problema" de *stack overflow* no és només característic de *gets*. Funcions com scanf també tenen aquest problema: en introduir una cadena no es comprova (per defecte) si la cadena és més llarga que el buffer on s'emmagatzema. Això pot ser aprofitat per un atacant per prendre control sobre el programa.

En compilar el codi `stack4.c` (a l'executable amb nom `stack4`) segurament us mostrarà un missatge d'avís que l'aplicació generada no és segura. Executeu i introduïu una cadena (de mida inferior a 64 bytes) i veureu que, un cop introduïda la cadena, us apareix la direcció de retorn. El programa s'executa correctament.

A la figura 3 es mostra l'estructura de la pila de `stack4`. El registre eip és on s'emmagatzema la direcció de retorn. És a dir, és la direcció de la següent instrucció a executar després de fer una crida a una funció (correspon, de fet, a la instrucció que hi ha després de la crida a la funció). Anem a canviar-la! Ho podem fer, atès que la funció *gets* ens ho permet fer: caldrà sobreescriure els valors

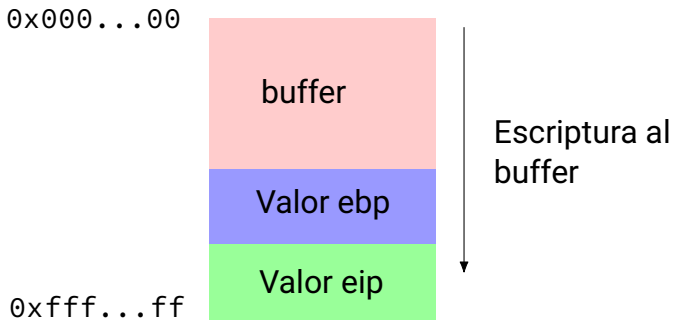

Figura 3: Estructura de la pila de `stack4`. La figura és del TFG d'Oriol Ornaque Blázquez titulat "Binary explotation".

de buffer[64], buffer[65], ... amb els valors adequats.

Executeu el següent codi

```
python stack4_exploit.py | ./stack4
```

Observeu la direcció de retorn que el programa mostra per pantalla. En analitzar el script de Python podreu veure que es pot controlar la direcció de retorn. Ara només fa falta adaptar-lo perquè retorni a la funció que ens interessa, *complete_level*. Per saber la direcció de memòria de la funció *complete_level* en executar-se l'aplicació feu el següent

```
readelf -s stack4 | grep complete_level
```

Us haurà d'aparèixer per pantalla una cosa similar a

```
64:  00000000004005e7 24 FUNC GLOBAL DEFAULT 14 complete_level
```

Ja està! Tenim la direcció de la funció *complete_level*. Editeu el fitxer `stack4_exploit.py` i modifiqueu l'script per introduir-hi la direcció de retorn de la funció *complete_level*. Després torneu a executar l'aplicació amb l'exploit de Python i us hauria d'aparèixer una cosa similar a

```
Welcome to phoenix/stack-four, brought to you by https://exploit.education
and will be returning to 0x4005e7
Congratulations, you've finished phoenix/stack-four :-) Well done!
```

Ho hem aconseguit! Hem pogut modificar la direcció de retorn de la funció!

Pregunta Quina és la direcció de retorn que us ha aparegut a vosaltres? Com heu modificat l'exploit de Python per modificar la direcció de retorn?

2.2 Injecció de codi

En el darrer exemple hem modificat la direcció de retorn per executar una altra funció. En un cas general volem aconseguir execució de codi arbitrari, no només tornar a les funcions ja presents al binari. Històricament aquest codi injectat s'ha anomenat *shellcode*. El nom shellcode es referia a injectar un programa shell que permet executar qualsevol altre ordre, però avui el terme s'usa de manera general per parlar de la injecció de codi maliciós. És possible programar un shellcode perquè faci qualsevol cosa que se'ns ocorri atès que en última instància és en si mateix un programa.

```c
#define LEVELNAME "phoenix/stack-five"

#define BANNER \
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

char *gets(char *);

void start_level() {
  char buffer[128];
  int *ret;

  printf("Buffer address: %p\n", buffer);
  gets(buffer);

  ret = __builtin_return_address(0);
  printf("and will be returning to %p\n", ret);
}

int main(int argc, char **argv) {
  printf("%s\n", BANNER);
  start_level();
}
```

Figura 4: Codi `stack5` de exploit.education. El codi original ha sigut modificat per mostrar la direcció de memòria del buffer per fer més fàcil fer proves amb el codi.

A la figura 4 es mostra l'exemple. La idea és "omplir" el buffer amb el codi màquina que ens interessa executar i fer que la direcció de retorn apunti cap a aquest codi màquina. Amb això aconseguim executar codi arbitrari.

Pregunta En aquest exemple particular, on ha d'apuntar la direcció de retorn per poder executar codi arbitrari?

Avui en dia, però, els sistemes operatius utilitzen tècniques de seguretat de programari per prevenir l'explotació de vulnerabilitats de corrupció de memòria. Quines tècniques utilitza avui en dia el sistema operatiu per dificultar aquests tipus d'atacs? Vegem-ho!

Compileu el codi i, en executar-lo, no introduïu cap cadena. Anem a analitzar el mapa de segments del procés que s'està executant. Per fer-ho, tal com es va veure a Sistemes Operatius 1, cal executar

```
cat /proc/<pid_proces>/maps
```

Aquesta instrucció ens permetrà veure on s'han mapat les zones de la pila, les llibreries dinàmiques que s'han executat, etc. Centrem-nos en la zona de la pila.

Executeu l'aplicació diverses vegades i observeu en quina direcció de memòria virtual s'emmagatzema la pila. Observeu també en quines direccions es mapen les llibreries dinàmiques que es carreguen.

Pregunta Què és el que observeu en executar diverses vegades la mateixa aplicació? On es mapen la pila així com les llibreries dinàmiques que es carreguen en executar-se l'aplicació?

Pregunta Es podrà executar codi màquina emmagatzemat a un buffer de la pila? Per què? A partir del mapa de la pila, quines conclusions podeu treure?

Pregunta A partir dels experiments anteriors, veieu factible ("senzill") fer la injecció de codi proposada? Raoneu la resposta.

Anem a "desactivar" temporalment les tècniques de seguretat per poder portar a terme l'experiment d'injecció de codi proposat. Per això obriu un terminal i executeu la següent comanda

```
setarch -R /bin/bash
```

Aquesta darrera comanda executa un nou shell... per tal de fer la injecció de codi caldrà executar l'aplicació fent servir fer aquest nou shell.

Pregunta Què és el que fa l'opció "-R" de la comanda? Per a què ens serà útil per fer la injecció de codi al buffer?

L'objectiu, com s'ha comentat abans, es omplir el buffer amb codi màquina executable i fer que la direcció de retorn apunti al principi d'aquest buffer. Podrem executar arbitrari! Però per poder-ho fer cal permetre que l'executable ens permeti executar codi que hi ha la pila. Per defecte no ens ho deixarà fer (per motius de seguretat, naturalment, per evitar que es pugui fer injecció de codi).

Anem a permetre que es pugui executar codi màquina que hi hagi a la pila. Per fer-ho fem

```
execstack -s stack5
```

Executeu diverses vegades el codi `stack5` i comproveu que la direcció de memòria en què s'emmagatzema el buffer és sempre la mateixa.

Editeu el fitxer `stack5_exploit.py` per tal d'introduir, a la direcció de retorn de la funció, la direcció que fa falta per tal de poder fer la injecció de codi. Ja ho tenim fet!

```
python stack5_exploit.py | ./stack5
```

En executar aquesta aplicació, assegureu-vos que a la direcció de retorn apareix la direcció on hi ha el codi injectat. En cas contrari no funcionarà....

Pregunta Què fa la injecció del codi que hem introduït?

Pregunta (Díficil) Quins bytes del codi injectat indiquen la instrucció a executar? Per tal de respondre a la pregunta, se us recomana revisar el codi assemblador associat a l'exercici així com fer servir un editor hexadecimal (per exemple, `ghex` o `okteta`) i veure en quins bytes s'emmagatzema la instrucció a executar. En respondre a la pregunta, comenteu el que heu trobat. Quina instrucció assemblador és la que conté el codi a executar?

Exercici (Difícil) Modifiqueu el codi injectat perquè executi una altra instrucció (de dues lletres com, per exemple, "`/bin/ps`" o "`/bin/df`") i comproveu que funciona. Com heu modificat el codi injectat? Quins bytes heu modificat?

Per acabar aquesta secció, executarem un codi injectat "sorpresa". Per executar aquest codi heu de procedir com abans: modifiqueu la direcció de retorn perquè apunti a la direcció on s'emmagatzemarà el codi injectat.

Es recomana tancar totes les aplicacions i tenir totes les dades desades a disc abans d'executar aquest codi.

```
python stack5_exploit_surprise.py | ./stack5
```

Pregunta Què ha fet el codi?

2.3 Return-oriented-programming

A la secció anterior hem vist un exemple d'injecció de codi. Per portar-ho a terme fa falta poder executar codi que hi ha a la pila. Actualment, però, els sistemes operatius configuren la pila de forma que no s'hi pugui executar codi. Assumim doncs un escenari on no és possible l'execució de codi emmagatzemat a la pila. Existeixen altres tècniques que permeten aprofitar les vulnerabilitats associades a un programa.

Com hem vist abans, podem executar codi existent modificant la direcció de retorn de la funció. El que es pot fer és que la direcció de retorn sigui la direcció d'una funció com pot ser una funció de la llibreria libc (*printf*, *fork*, ...). El fet de poder executar una funció de la llibreria libc permet executar, per exemple, funcions molt útils com l'intèrpret de comandes. Aquesta tàctica es denomina return-to-libc ja que el codi utilitzat per vulnerar el programa són funcions d'aquesta llibreria.

El retorn a libc és part d'un tipus d'atac més ampli que s'anomena Return Oriented Programming (o ROP). En termes generals l'estratègia de ROP tracta de concatenar seqüències d'instruccions ja existents en el programa vulnerable, denominats gadgets. Aquests gadgets, en l'ésser executats, aconsegueixen el comportament desitjat per l'atacant, modificar l'estat dels registres i realitzar crides a sistema que permetin prendre el control de el programa vulnerable. Quan aquestes instruccions són part de libc és que estem davant d'un atac del tipus return-to-libc.

En aquesta pràctica no s'implementarà aquest tipus d'atac.

3 Heap overflow

L'emmagatzematge dinàmic és un terme comú per referir-se a una part de memòria a la qual els programes poden assignar memòria en temps d'execució per a objectes de mida molt gran o desconeguda en el moment de la compilació, i per tant, el compilador no els pot gestionar a la pila. Aquesta porció de memòria és sovint gestionada per biblioteques anomenades assignadors, encarregades de portar un registre de memòria assignada i alliberada, la seva mida, la possibilitat de reutilitzar trossos alliberats i la fusió d'alliberats trossos per evitar la fragmentació d'aquesta.

L'estàndard C inclou definicions per a un conjunt de funcions d'emmagatzematge dinàmic comunes a tothom però la implementació depèn del sistema operatiu i de la biblioteca. Algunes de les funcions que típicament s'utilitzen més són el *malloc*, calloc, realloc i free. La biblioteca GNU C *malloc* conté implementacions per a aquestes funcions. Observar que aquestes funcions són funcions de la llibreria libc. És a dir, les funcions com *malloc*, *calloc*, *realloc* i *free* són funcions que formen part d'una llibreria d'usuari. Es pot treure profit de la implementació d'aquestes funcions per fer un atac.

La signatura de la funció *malloc*, memory allocation, de la llibreria libc de Linux és la següent: *void *malloc(size_t size)*. Com a paràmetre d'entrada rep un nombre el bytes a reservar (un sencer de 64 bits sense signe) i retorna un apuntador al bloc de dades que s'ha reservat. Una forma d'implementar el *malloc* és fent servir la crida a sistema *sbrk*, una funció que permet manipular l'espai de heap del procés (la heap és la porció de memòria assignada per fer reservar memòria de forma dinàmica).

La funció *sbrk* es pot interpretar intuïtivament com una funció que permet augmentar o disminuir la quantitat d'aigua (i.e. memòria dinàmica) associada al un pantà (i.e. procès), veure figura 5. La crida *sbrk(0)* retorna el nivell de l'aigua actual del pantà (i.e. un apuntador al nivell actual del heap). Si hi especifiquem qualsevol altre quantitat com a paràmetre podem augmentar o disminuir

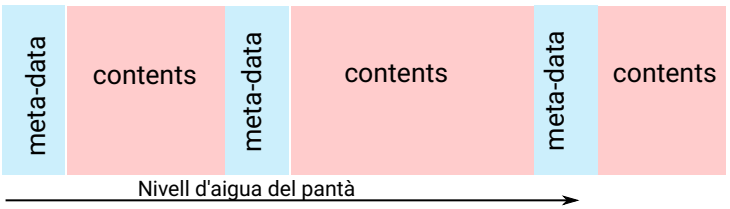

Figura 5: La llibreria libc de GNU utilitza la funció *sbrk* per gestionar la memòria dinàmica. Intuïtivament funciona de forma similar a una pila.

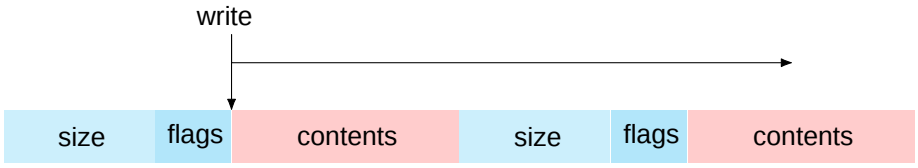

Figura 6: En fer servir la funció strcpy es pot produir un overflow, cosa que un atacant pot aprofitar per prendre control sobre la màquina. La figura és del TFG d'Oriol Ornaque Blázquez titulat "Binary explotation".

el nivell de l'aigua del pantà, i.e. el heap s'incrementa o disminueix en aquest valor i la funció retorna un apuntador al valor antic abans de fer la crida.

Així, per exemple, la crida *sbrk(1000)* augmenta en 1000 bytes el heap i retorna un apuntador a l'inici d'aquests 1000 bytes de forma que es puguin fer servir els 1000 bytes per l'aplicació. Observar, en canvi, que si es fa la crida sbrk(-1000) es disminueix en 1000 bytes l'espai de memòria associat a la heap, el valor retornat és un apuntador al nivell de heap abans de fer la crida. La funció *sbrk(size)* retorna un -1 en cas que no s'hagi pogut realitzar l'operació desitjada.

La funció *malloc*, que internament utilitza la crida a sistema *sbrk,* permet gestionar els blocs de memòria dinàmica assignats a un programa amb un estil similar a la pila, veure figura 5. Aquesta característica permet que un atacant ho aprofiti per prendre control sobre la màquina. A la figura 6 es mostra la idea bàsica: com es pot veure, escriure fora de l'espai reservat per a les variables permet que s'escrigui en zones en què no està previst escriure-hi.

3.1 Sobreescriptura de la direcció de retorn

Anem a aprofitar aquesta característica per fer modificar el flux d'execució d'un programa tal com hem fet a la secció 2.1. A la figura 7 es mostra el codi `heapone` que s'ha obtingut de `https://exploit.education/`. L'objectiu és aconseguir fer cridar la funció *winner* així que s'acabi l'execució de la funció *start_ level*. Dit d'altra forma, es tracta de sobreescriure la direcció de retorn de la funció amb la direcció on es troba winner. Això es pot fer aprofitant la "debilitat" de la funció *strcpy*, que permet copiar cadenes de qualsevol longitud. Al manual del *strcpy* ja ho indica: "If the destination string of a strcpy() is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine."

A la figura 8 es mostra l'estructura del heap un cop s'ha reservat memòria per a les dues estructures. S'hi mostra també l'objectiu a aconseguir: amb el primer *strcpy* se sobreescriu el valor de *i2->name* (i.e. *&name* a la figura): s'hi escriurà la direcció en què s'emmagatzema la direcció de retorn de la funció. En fer el segon *strcpy*, la cadena origen a copiar (i.e. *argv2*) contindrà la nova

```c
struct heapStructure {
    int priority;
    char *name;
};

void winner() {
    printf("Congratulations, you've completed this level!!!\n");
    exit(0);
}

void start_level(char *argv1, char *argv2)
{
    struct heapStructure *i1, *i2;
    unsigned long int value;
    void *pointer;

    i1 = malloc(sizeof(struct heapStructure));
    i1->priority = 1;
    i1->name = malloc(8);

    i2 = malloc(sizeof(struct heapStructure));
    i2->priority = 2;
    i2->name = malloc(8);

    pointer = __builtin_return_address(0);
    printf("Original return address: %p\n", pointer);

    strcpy(i1->name, argv1);

    pointer = (void *) i2->name;
    printf("A i2->name s'emmagatzema el valor %p\n", pointer);

    strcpy(i2->name, argv2);

    pointer = __builtin_return_address(0);
    printf("New return address: %p\n", pointer);

    printf("and that's a wrap folks!\n");
}


int main(int argc, char **argv) {
    printf("%s\n", BANNER);
    if (argc == 3)
        start_level(argv[1], argv[2]);
    else
        printf("Need two arguments\n");
}

```

Figura 7: Codi `heapone` de exploit.education adaptat per a la pràctica. En aquest exercici l'objectiu se centra en aconseguir cridar la funció winner de forma similar a com es fa fer al codi `stack4`.

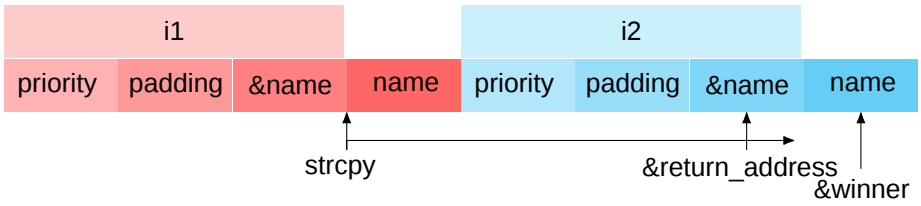

Figura 8: Estructura del heap un cop s'ha reservat la memòria dinàmica per a les dues estructures. La figura mostra també quin és l'objectiu a aconseguir (veure text). La figura és del TFG d'Oriol Ornaque Blázquez titulat "Binary explotation".

direcció de retorn (i.e. la direcció de *winner*). El resultat final és que se sobreescriurà la direcció de retorn amb la direcció de winner. Així que se surti de la funció, s'executarà la funció *winner*!

Per tal d'aconseguir-ho es proporciona un codi de `heapone.c` que imprimeix per pantalla múltiple informació per facilitar arribar a l'objectiu. Per aconseguir-ho es proporciona un script, `heapone_exploit.sh`, que permet generar els dos paràmetres que es passaran com a argument a `heapone` en executar-lo.

Amb el que s'ha fet a la secció 2 tenim totes les eines per aconseguir-ho:

Pregunta Quin és el valor que s'haurà d'assignar a a *i2->name*? Com aconseguiu obtenir aquest valor? Detalleu la resposta.

Pregunta Quin és el contingut que s'haurà d'escriure a *i2->name*? Com aconseguiu obtenir aquest valor? Detalleu la resposta.

Pregunta Fa falta activar el bit perquè la pila pugui contenir codi executable? Raoneu la resposta.

Pregunta Com construïu l'script a executar? Quin valor assigneu a A? Quin valor assigneu a B?

3.2 Un cas real: sudo

Sudo és una potent utilitat que s'inclou a la majoria, si no a tots, els sistemes operatius basats en Unix i Linux. Permet als usuaris executar programes amb privilegis de seguretat d'un altre usuari. A finals de gener del 2021 l'equip de recerca Qualys va revelar una vulnerabilitat a la comanda sudo que permetia una escalada de privilegis mitjançant heap overflow. Aquesta vulnerabilitat va ser introduïda al codi de sudo el juliol del 2011 i ha estat amagada més de 10 anys. L'explotació amb èxit d'aquesta vulnerabilitat permetia a qualsevol usuari sense privilegis obtenir privilegis d'administrador a l'amfitrió vulnerable.

Per a més informació consulteu https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit. En aquest enllaç es mostra un vídeo en què un usuari sense privilegis guanya privilegis d'administrador a la distribució d'Ubuntu 20.04: https://vimeo.com/504872555.

4 Entrega de la pràctica

Es demana **entregar un fitxer ZIP** que inclogui les respostes a les preguntes realitzades a les seccions 2 i 3. El nom del fitxer ha d'indicar el número del grup, seguit del nom i cognom dels

integrants del grup (e.g. **P2_Grup07_nom1_cognom1_nom2_cognom2.zip**). L'informe a entregar ha d'estar en **format PDF** o equivalent (no s'admeten formats com `odt`, `docx`, ...).

L'informe estarà estructurat en una introducció, treball realitzat i conclusions. A la part d'introducció es descriu breument el problema solucionat (i.e. resum del que proposa en aquesta pràctica). A la part de del treball realitzat es responen les preguntes realitzades a la pràctica i es descriuen les proves que s'han realitzat. Per respondre les preguntes podeu seguir un fil conductor que us permeti descriure el treball realitzat. També podeu respondre de forma explícita a cadascuna de les preguntes (en aquest cas indiqueu clarament a quina pregunta esteu responent en cada moment). Sigueu breus i clars en els comentaris i experiments realitzats, no cal que us esteneu en el text. Finalment, a la part de conclusions es descriuen unes conclusions tècniques de les proves realitzades. Per acabar, es poden incloure conclusions personals.

Inclogueu, preferentment en format text, les comandes que heu executat i algun comentari breu descrivint el resultat si ho creieu necessari. En cas que preferiu incloure captures de pantalla en comptes d'incloure el resultat en format text, assegureu-vos que el text de la captura es pot llegir bé (és a dir, que tingui una mida similar a la resta del text del document) i que totes les captures siguin uniformes (és a dir, que totes les captures tinguin la mateixa mida de text).

El document ha de tenir una **llargada màxima de 5 pàgines** (sense incloure la portada). El document s'avaluarà amb els següents pesos: proves realitzades i comentaris associats, un 60%; escriptura sense faltes d'ortografia i/o expressió, un 20%; paginació del document feta de forma neta i uniforme, un 20%.