Introducció

José María Alonso Cebrián
Jordi Gay Sensat
Antonio Guzmán Sacristán
Pedro Laguna Durán
Alejandro Martín Bailón
Jordi Serra Ruiz

PID_00208378

Universitat Oberta de Catalunya

Índex

Objectius..... 5

1. Introducció..... 7
1.1. *Bug*..... 8
1.2. *Exploits*..... 10
1.2.1. Què és un *exploit*? 10
1.2.2. Tipus d'*exploits*..... 11
1.2.3. Creació d'*exploits*..... 12
1.2.4. Memòria del procés 12
1.2.5. El mercat de l'*exploit*..... 13
1.2.6. Valoració de l'*exploit*..... 16
1.2.7. Estructura de l'*exploit*..... 16

2. Gestió de memòria..... 18
2.1. Ordre d'escriptura de bits i bytes en memòria 18
2.2. Segments 19
2.3. Registres bàsics per a l'execució de programes 20
2.3.1. Registres de propòsit general 21
2.3.2. Registres de segments 23
2.3.3. Registre EFLAGS 24
2.4. Utilització de la pila 27

3. Conceptes bàsics de llenguatge màquina..... 30
3.1. Tipus de nomenclatures 31
3.2. Operacions i operands en assemblador 32
3.3. Adreçament i operands 34
3.4. Desassemblament d'un petit programa 37
3.4.1. Entorn: Linux + GCC 37
3.4.2. Entorn: Windows + BorlandC (BCC) 40
3.4.3. Comparativa dels entorns 42

Objectius

En finalitzar la lectura d'aquest material, els estudiants hauran aconseguit les competències següents:

1. Conèixer el funcionament de la programació a baix nivell.
2. Conèixer la funció dels registres del sistema.
3. Comprendre com funciona un *exploit*.
4. Saber fer un *exploit* sobre un sistema real.
5. Conèixer les tècniques de defensa contra els *exploits* de programació.
6. Saber utilitzar les eines existents per al seguiment de programari.
7. Saber canviar el comportament de programari en execució.

1. Introducció

Les màquines no s'equivoquen, ens equivoquem nosaltres. No és una visió tenebrista i negativa de l'ésser humà, sinó que, com ja va dir sant Tomàs d'Aquino fa molt temps, "res imperfecte no pot generar res perfecte". Cregueu o no en les tesis de sant Tomàs, la veritat és que els sistemes operatius, les aplicacions i, en definitiva, el programari en general, tenen errors.

Aquests errors en el programari es poden produir per un mal disseny de l'arquitectura, una mala comprovació de les premisses d'entorn per a les quals va ser creat o simplement per una implementació errònia de les especificacions.

Un programari que funcioni serà aquell que faci exactament tot allò per a què va ser creat i dissenyat. No obstant això, el programa pot ser funcional però insegur. Un programari segur serà aquell que fa allò per a què va ser creat i res més.

Aquesta cosa més que aconsegueix fer el torna vulnerable i el converteix en un risc per als pilars de la seguretat d'aquelles màquines o entorns on estigui corrent aquest programari.

Suposem un programa que rep dos nombres com a paràmetres d'entrada per a multiplicar-los. Un programa funcional podria ser aquell que reculli els dos nombres, els emmagatzemi en dues variables, invoqui la funció multiplicar i retorni el resultat en una tercera variable. Aquesta pot ser la manera com molts han après a programar, però quan parlem de programari segur, aquesta manera no és correcta. Un programa segur seria aquell que comprova exactament la mida de les dades d'entrada, comprova que són dos nombres, s'assegura que són dos nombres del tipus de dades de les variables que s'han reservat i que tots dos no desborden el tipus de dades de la variable que emmagatzemarà el resultat.

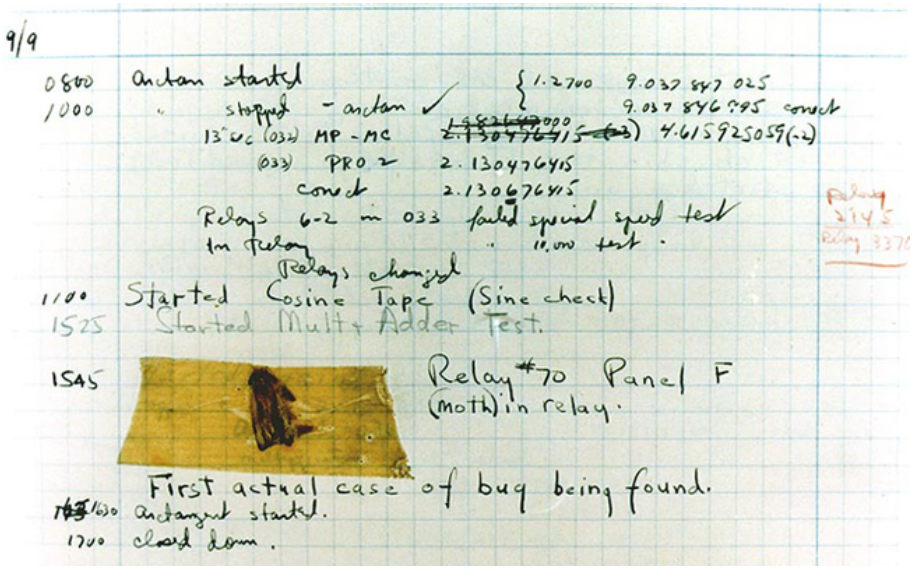
Els límits i premisses que s'han de comprovar abans de crear un programari són molts i, si això no es compleix, llavors tindrem un programari insegur.

Durant aquest curs repassarem els errors més comuns i com es poden explotar per a obtenir "aquesta funcionalitat més" que permeti a un atacant treure profit de la vulnerabilitat.

1.1. *Bug*

Si retrocedim fins als inicis de la computació com a ciència, trobem ja els primers problemes amb els programes. Encara que ja s'havien trobat errors en els programes anteriorment, va ser l'any 1947, investigant per què un programa dels primers ordinadors, el *Mark II*, funcionava de manera errònia, quan van descobrir que el mal funcionament es devia a una arna que hi havia en un dels relés electromagnètics.

Grace Murray, programadora del *Mark II*, va recollir i va enganxar l'insecte amb cinta adhesiva en un paper i s'hi va referir com el *bug* ('bestiola', en anglès) que causava el problema. Aquest va ser el primer cas en què el terme *bug* es referia explícitament a errors en un programa.



Es coneixen com a *bugs*, per tant, tots i cadascun dels errors de programació que es detecten en un programari.

No tots els errors són iguals, ni de bon tros, i per això cal catalogar-los depenent de molts factors. Per a conèixer correctament totes les implicacions d'un error es generen les bases de dades d'expedients de seguretat, en què els errors seran estudiats en profunditat per la comunitat tècnica o de seguretat.

De la mateixa manera, no totes les bases de dades d'expedients de seguretat són iguals, ni tenen tots els errors, ni tots els errors acaben en bases de dades d'expedients de seguretat. Sempre hi haurà persones que trobin vulnerabilitats (o errors) i no les reportin a la comunitat, sinó que ho faran a les màfies, que arriben a pagar grans quantitats de diners per això.

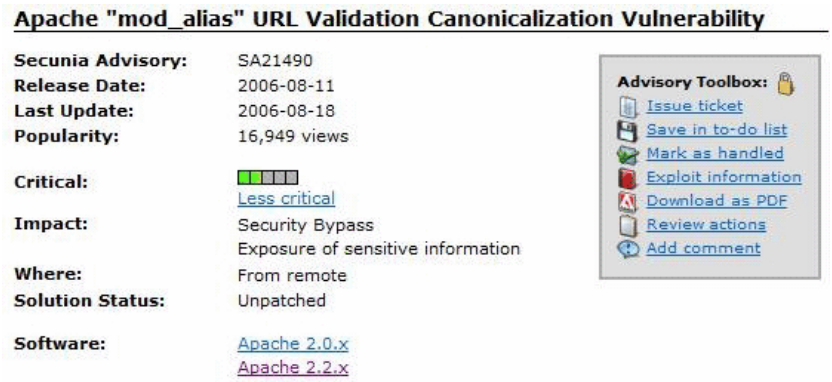
El volum de programari i la rapidesa de les versions de cada programa fan que documentar tots els errors de totes les versions de tots els programes sigui una tasca inabordable. En altres situacions, l'error es produeix en codi i per motius de protecció de l'avantatge competitiu, la companyia desenvolupadora del programari en limita, en la mesura que pot, la informació.

Altres errors simplement es troben en programari privat que només una empresa utilitza perquè ha estat creat expressament per a aquesta, és a dir, és un programari a mida, i tret que l'empresa mantingui una base de dades privada d'expedients de seguretat, aquests no s'emmagatzemen en cap altre lloc.

No obstant això, les bases de dades d'expedients de seguretat són de gran utilitat i disposen avui dia d'un enorme volum d'errors documentats.

Secunia

Per posar un exemple, *Secunia*, que manté una de les bases de dades d'expedients de seguretat més respectades per la comunitat tècnica dedicada a seguretat informàtica, recull, avui dia, errors de més de 20.500 productes de més de 2.300 fabricants de programari.



Error en programari *mod_alias* d'Apache 2.2.x emmagatzemat a la base d'expedients de seguretat en Secunia. Font: http://secunia.com/advisories/21490/

Pàgines web

Avui dia hi ha múltiples bases de dades, però potser les més importants es podrien resumir en la llista següent:

Secunia: http://secunia.com/advisories/

SecurityFocus: http://www.securityfocus.com/bid

Common Vulnerabilities and Exposures: http://cve.mitre.org/cve/

En totes, com es pot veure en la figura anterior, cada error rep un número d'identificació únic. En l'exemple de l'error d'Apache aquest número és l'SA21490, però aquest mateix error pot tenir altres identificadors en altres bases de dades. En **Security Focus**, aquest número es coneix com a *BugtraqID* i en **Common Vulnerabilites and Exposures** és conegut, simplement, com a CVE.

Aquest codi identifica de manera unívoca un error en una base de dades d'expedients de seguretat i a partir d'aquest punt, l'error de seguretat comença a ser estudiat per a conèixer-ne les implicacions de seguretat.

1.2. *Exploits*

1.2.1. Què és un *exploit*?

L'*exploit* és un programa creat especialment per a treure partit d'un error. Aquests *exploits* poden ser utilitzats per a extreure informació d'una base de dades, per a deixar sense connexions un servidor web que no gestiona bé l'assignació de recursos o fins i tot per a executar un codi arbitrari en la màquina on el programari vulnerable està essent executat.

Poder executar un codi arbitrari significaria que un atacant podria aconseguir l'execució del codi o el programa que volgués, com per exemple una simple calculadora en Windows (molt usada en les proves de concepte perquè és innòcua).

Ens centrarem en les maneres com els *exploits* poden ser creats per a saber fins a on poden ser utilitzats. Com ja hem dit, aquests *exploits* podran ser utilitzats per a elevar privilegis en la màquina, extreure informació de la base de dades, fer que un sistema deixi de funcionar o executar un codi remot en una màquina per mitjà d'Internet. Tots i cadascun tindran importància en el seu entorn.

Exploit local

Un exemple famós d'*exploit* local és el que va permetre executar codi en *la consola Wii*. Aquesta consola està tancada per defecte per a l'execució només dels jocs autoritzats. L'*exploit* va ser publicat el juny del 2007 i és conegut com el *Twilight Hack*, perquè l'error es produeix en el joc *The Legend of Zelda: Twilight Princess*. En aquest joc el protagonista està acompanyat d'un cavall anomenat *Epona*, detall que no tindria més importància si no fos perquè aquest nom apareix dins dels fitxers de les partides desades. Els membres del grup *Team Twiizers* van descobrir que si canviaven el nom del cavall per un d'especialment modificat, podien executar els seus propis jocs en la Wii. Aquest *exploit* permet, a tot aquell que tingui el joc *The Legend of Zelda: Twilight Princess*, executar qualsevol programa en la consola Wii.

Exploits remots

Altres exemples d'*exploits*, en aquest cas remots, són els que van afectar diferents productes de Microsoft a l'inici del segle XXI. *Exploits* com *Code Red* o *SQL Slammer*, tots dos cucs que s'aprofitaven d'errors explotables remotament, es van fer molt famosos pel nombre de màquines que van afectar.

Cuc

Un cuc és un virus que, a més d'infectar el sistema i fer una sèrie d'accions, intenta infectar altres equips que es trobin a prop mitjançant la mateixa vulnerabilitat. Poden arribar a produir efectes devastadors si l'error del qual s'aprofiten és un error per al qual no hi ha un pegat durant llarg temps.

El primer utilitzava una vulnerabilitat en el servei *IIS* per a aconseguir executar codi en memòria i el segon s'aprofita d'un error en la configuració per defecte del compte *sa*, amb privilegis administratius, en els servidors SQL Server.

Podem trobar desenes d'*exploits* que aprofiten els errors de programació del lector d'arxius PDF. Per la seva gran difusió, s'està tornant el centre d'atenció d'aquells que desenvolupen *exploits* per a poder aconseguir el control de les màquines en què s'executen. Podem trobar fitxers aparentment normals en

format PDF que han estat modificats expressament per a explotar alguna de les vulnerabilitats del lector d'Adobe. L'abril del 2009 Adobe reconeix un nou *0-day* de l'Acrobat, una vulnerabilitat molt greu de la qual se saben tots els detalls. Com que és *0-day*, l'*exploit* és públic, per la qual cosa es podia explotar i executar codi en el sistema. L'error estava en la funció *getAnnots* i permet a un atacant, de manera senzilla, executar codi en el sistema de manera remota. Creant un document PDF amb una anotació i afegint un *script* al PDF per mitjà de la funció *OpenAction*, ja tenia accés a executar codi arbitrari.

El mateix ocorre en algunes versions del tractament de textos MS Word, en el qual es pot executar codi en les macros.

El programari lliure tampoc no se salva d'aquests errors; són coneguts els problemes de seguretat que hi ha hagut en els programes que s'alliberen amb llicències lliures.

Tot el programari està exposat a tenir errors, ja que els desenvolupadors poden cometre errors de programació per distracció. És molt fàcil deixar de fer alguna comprovació en una de les milers o milions de línies de codi d'un programa.

1.2.2. Tipus d'*exploits*

Dins dels *exploits* cal diferenciar entre els *exploits* de dia zero i la resta. Un *exploit* de dia zero és aquell per al qual no hi ha un pegat creat que solucioni el problema.

El cicle normal d'una vulnerabilitat comença amb el descobriment de l'error. Si aquest error és descobert per la companyia propietària del programari, és probable que no surti a la llum fins que estigui disponible un pegat que arregli el programari. Això és molt comú en les grans empreses de programari com Microsoft, en què s'aplica el principi de *responsible disclosure*, és a dir, no es publica res que pugui afectar la seguretat dels seus clients.

En els entorns en els quals no es coneix l'error fins que surt el pegat, els investigadors de seguretat fan processos d'enginyeria inversa sobre el pegat per a descobrir l'error que arregla. Així, veient el que arregla saben el que està malament i generen un *exploit* que tregui partit d'aquest error en els equips que no tinguin el pegat aplicat.

En els entorns de *full disclosure*, l'error es fa públic a Internet al mateix temps que es descobreix i la cursa entre el fabricant de programari i els creadors de l'*exploit* pot ser guanyada per l'un o l'altre. Alguns anomenen aquests *exploits* de dia 1, ja que es coneix l'error i no hi ha pegat, però encara no hi ha *exploit*.

Els *exploits* de dia zero són els més perillosos per a la indústria del programari. Es tracta d'errors de seguretat que només coneixen una o diverses persones i que es poden explotar en tots els sistemes, ja que el fabricant del programari

desconeix aquesta vulnerabilitat. Aquests són els més cotitzats i els que miren de ser captats per la indústria del programari maliciós i les empreses de seguretat.

1.2.3. Creació d'*exploits*

Aquest és l'objectiu del curs, comprendre com un simple nom d'un cavall pot provocar un error de seguretat en un sistema, conèixer quines són les tècniques per a la generació d'*exploits* i quins són els exemples més típics quan parlem de programari de plataforma.

Com a introducció a la creació d'*exploits*, entendrem primer com treballa internament un programa en l'ordinador, com emmagatzema de manera bàsica els valors que maneja un programa i també com una mala comprovació de les dades d'entrada pot generar un comportament insegur del programa.

1.2.4. Memòria del procés

En la creació d'un programa en un llenguatge d'alt nivell, per exemple en llenguatge C, es fa un ús intensiu de crides a funcions. Aquestes funcions permeten al programador abstreure's de la tasca (per exemple, per a fer que un text sigui imprès per pantalla, amb la simple invocació de la funció *printf* aconsegueix aquesta acció). Cadascuna de les crides a aquestes funcions genera la creació d'un subprocés, el pas de paràmetres entre el procés principal del programa que crida i el subprocés cridat, el lliurament del control del programa principal a la funció cridada, l'execució del codi de la funció, el lliurament de resultats del subprocés cridat al procés principal i el retorn del control d'execució al procés que va invocar la funció.

És a dir, una simple crida com `printf("Hola")` generaria la creació d'un nou procés amb el codi de la funció *`printf`*, el traspàs de l'adreça del procés principal, on la funció `printf` ha de retornar el control una vegada acabada l'execució, la creació d'un espai de memòria per a passar els paràmetres (en aquest cas la cadena ASCII `Hola`) i el traspàs del control d'execució a l'adreça on comença el codi de la funció.

La funció accedirà als paràmetres, executarà el codi, crearà un valor de resposta de la funció (que en aquest cas és un codi d'estat d'error) i una vegada acabada l'execució retornarà el control a l'adreça de memòria que li ha deixat el procés principal. Tot aquest intercanvi de dades es farà en la memòria, dins d'unes estructures especials.

Quan s'executa un procés dins d'un sistema operatiu, aquest tindrà dues parts diferents en memòria. Una part estarà formada per variables i valors que el sistema configura en la creació del procés i l'altra serà la zona de memòria pròpia, que s'utilitzarà per a desar les variables globals i les variables locals. Serà dins d'aquesta zona on es crearan els subprocessos.

Cada subprocés creat per la invocació d'una funció tindrà una zona de memòria pròpia anomenada *pila* (*stack*). Aquesta zona de memòria serà accessible tant pel procés principal com pel subprocés. Aquesta zona de memòria rep el nom de *pila* pel sistema de gestió que utilitza, basat en la filosofia LIFO (*last in, first out*), o el que és el mateix, l'últim que entra és el primer que surt.

Com es veurà, no comprovar la mida dels valors abans de ser assignats a les variables pot permetre que se sobreescriguin valors de variables. Això donarà lloc no solament a la sobreescriptura de variables, sinó també la introducció de codi i la sobreescriptura de punters d'execució, i permetrà arribar a executar codi arbitrari en la màquina.

En l'exemple anterior de l'*exploit* local per a executar codi en la consola Wii es va utilitzar una cosa similar amb el nom del cavall *Epona*. L'*exploit* canvia el nom del cavall per un valor que sobreescriu la memòria de la consola de tal manera que permet cridar qualsevol programa que es vulgui.

I és possible en tots els programes, això? No, rotundament no. Aquests errors són comuns en llenguatges de programació com C i C++, llenguatges que no incorporen per defecte cap tipus de protecció enfront de la sobreescriptura de la memòria. Hi ha, això sí, funcions segures en els llenguatges C i C++ que comproven la mida de les variables i que, en cas d'usar-se, evitarien els errors de desbordament.

A més d'aquestes proteccions veurem que els llenguatges, els sistemes operatius i els compiladors més moderns implementen mecanismes de protecció contra aquest tipus de vulnerabilitats. Veurem totes aquestes proteccions en els últims mòduls.

1.2.5. El mercat de l'*exploit*

L'evolució en els sistemes de seguretat informàtics, com és ben sabut, ha arribat a tots els àmbits, inclòs el del sector comercial. Enrere van quedant els dies en els quals tota la informació era considerada pública, inclosa la concernent a la seguretat. Els investigadors feien públics els seus treballs pel simple fet del seu reconeixement general. En canvi, avui dia, per a alguns investigadors de vulnerabilitats el lucre i els diners són motivacions tan importants o fins i tot més que el reconeixement d'aparèixer en un lloc o un altre per haver descobert un error en un programari.

Entorn del sector de la seguretat de les vulnerabilitats i els *exploits* s'ha generat un mercat de compravenda bastant significatiu, amb xifres que de vegades maregen, i en el qual els productes es cotitzen de la mateixa manera que un valor en una borsa qualsevol.

El descobriment d'un error de seguretat i la sortida del programa consegüent que l'aprofita obren nombroses perspectives per a utilitzar-lo: *spammers*, timadors, extorsionadors o espies tecnològics semblen formar un col·lectiu summament interessat en l'adquisició d'aquestes peces tecnològiques. L'ús de xarxes de zombis o la presa de control de determinades màquines per a fer un atac i després eliminar qualsevol prova representen una temptació molt interessant que conviden a invertir en la tecnologia de les vulnerabilitats.

Programari zombi

Com a programari zombi es coneix el conjunt de programari que s'instal·la en l'ordinador mitjançant algun cavall de Troia i que permet controlar l'ordinador remotament. És utilitzat per a enviar correu brossa (*spam*) o per a fer atacs massius contra servidors.

No obstant això, conscients de la necessitat de premiar econòmicament els descobridors dels errors importants, les empreses del sector de la seguretat fan sonats pagaments per aconseguir el tan preuat botí i evitar els atacs abans que apareguin. Una empresa pot licitar per la compra d'un dels paquets en venda pensant en millores dels seus sistemes d'auditoria, el seu posicionament en el mercat o simplement moguda per temes de competència.

Mercat negre d'*exploits*

El negoci de la venda dels *exploits*

Un exemple bastant interessant del lucratiu negoci de la venda d'*exploits* va ser el que va treure a la llum Trend-Micro l'any 2006. Analistes d'aquesta companyia, que es van infiltrar al món *underground* de les xarxes *hacker*, van detectar la venda d'un *exploit* per a Windows Vista per la no gens menyspreable xifra de 50.000 dòlars. Un any abans, la companyia Kaspersky feia pública també la possible venda d'una vulnerabilitat de dia zero per al format de fitxer WMF per part d'un col·lectiu *hacker* rus al preu de 4.000 dòlars. Aquesta vulnerabilitat permetia l'execució remota de codi per mitjà del navegador Intenet Explorer.

Encara que les xifres esmentades aquí poden semblar significativament altes, no són ni més ni menys que una apreciació real del mercat negre que es mou en aquest sector i els diners que genera actualment el mercat del programari maliciós. Evidentment, la procedència dels diners no és tan legal com un desitjaria, i com a tal, tot aquest negoci es troba fora de l'empara legal establerta.

Mercat d'empreses de seguretat

Per contra, hi ha un altre tipus de mercat més lícit per a evitar aquestes pràctiques irregulars, com les iniciatives promogudes per les empreses de seguretat informàtica. Incentiven la publicació d'errors, i ofereixen una recompensa econòmica al fruit de tota una investigació. En aquest sentit podem trobar la iniciativa Zero Day Initiative (ZDI), fundada per TippingPoint. Consisteix en un programa per a recompensar els investigadors de seguretat pel fet de divulgar vulnerabilitats responsablement. Els beneficis econòmics obtinguts

per mitjà d'aquesta iniciativa no són tan alts com els que es poden aconseguir en el mercat negre, però els diners guanyats així són totalment legals i com a tals són lliures de qualsevol complicació.

Beneficis del programa Zero Day Initiative

Zero Day Initiative

Per mitjà de l'URL http://www.zerodayinitiative.com/about/benefits/ es poden conèixer quins són els beneficis que es poden obtenir pel fet de donar a conèixer una vulnerabilitat descoberta. Els errors descoberts són valorats i quantificats mitjançant una sèrie de paràmetres que mesuren la criticitat, l'impacte o la reproductibilitat de l'atac. Proporcionen una sèrie de punts que poden ser canviats per diners en metàl·lic o usats per a inscriure's i viatjar a conferències de reconegut prestigi en el sector de la seguretat, com per exemple la DEFCON i la BlackHat.

iDefense Labs

Per mitjà del seu programa VCP (Vulnerability Contributor Program), que va començar el seu camí l'any 2002, també incentiven econòmicament la publicació d'errors de seguretat i les proves de concepte d'aquests amb un valor màxim de 15.000 dòlars. A més, anualment presenten un certamen en què les vulnerabilitats del programa més significatives de l'any són premiades amb quanties que van des dels 5.000 fins als 50.000 dòlars.

Pack Agora

L'empresa Gleg Ltd. va adquirir l'any 2007 un paquet d'*exploits* de dia zero generats per Argeniss, dins dels quals destacava l'explotació de bases de dades Oracle per mitjà d'un dels elements del paquet adquirit. Aquest paquet, juntament amb altres mòduls de Gleg Ltd., va ser comercialitzat l'any 2008 en el Pack Agora, que es podia comprar per a fer auditories i tests de seguretat.

WabiSabiLabi

Un altre exemple interessant d'aquest mercat emergent el va proposar la firma suïssa WabiSabiLabi, que l'any 2007, sobre la base d'un estudi de mercat que va fer, va decidir plantejar un sistema similar a eBay però dins del mercat de les vulnerabilitats. El seu periple va començar amb vulnerabilitats per a Yahoo Messenger i Squirrelmail, però en aparença l'estudi de mercat no va ser totalment satisfactori o bé la idea no va ser gaire ben plantejada, ja que actualment el servei ja no està disponible.

1.2.6. Valoració de l'*exploit*

Dins del mercat de compravenda, independentment de com de legal o il·legal pugui ser, el preu que pot assolir una vulnerabilitat o un *exploit* està prefixat per una sèrie de paràmetres, entre els quals la popularitat de l'aplicació o el sistema afectat és el més important. Com tothom reconeixerà, un error de seguretat en l'Adobe Reader té més valor en el mercat que el detectat en una aplicació de poc ús o de difícil repercussió. Així mateix, no tindrà la mateixa graduació una vulnerabilitat d'un sistema operatiu modern, com per exemple el Windows Vista, com la que pot oferir actualment un error de seguretat del Windows 2000.

Un altre dels factors importants d'un *exploit* és el nivell d'aplicació que es pot arribar a oferir o la facilitat d'ús d'aquest. No és el mateix un *exploit* que permeti l'execució remota de codi amb elevació de privilegis que un altre en el qual l'explotació calgui fer-la localment i que per a això, a més, ja s'hagin de tenir determinats privilegis adquirits.

Avui dia tot té un preu, encara que la quantificació es revalora o cau depenent del moment. No obstant això, determinats elements es poden considerar de manera constant com els *top* dins dels més pagats i, és clar, dels més demanats. Errors de seguretat en serveis arxiconeguts com Hotmail, Gmail o Messenger, els últims sistemes operatius de Microsoft o aplicacions tan comunes com l'Adobe Reader, l'Adobe Flash i els servidors Apache, constitueixen la punta de llança d'un "sector comercial" en constant auge.

1.2.7. Estructura de l'*exploit*

En aquest llibre mostrarem dues parts ben diferenciades. La primera estarà centrada en les eines i els conceptes necessaris per a poder entendre la segona, que se centrarà més en els diferents *exploits* i vulnerabilitats dels sistemes.

Començarem, llavors, amb una visió no gaire detallada del llenguatge màquina i de l'assemblador, que ens permetrà comprendre com es pot variar segons la nostra voluntat el comportament d'un programari ja existent.

En aquesta part cobren molta importància els registres interns de l'ordinador, ja que sobre aquests aniran les adreces de memòria que podrem variar *a posteriori*. Veurem, doncs, els diferents registres i què s'hi desa usualment.

A continuació veurem la descripció dels operands d'assemblador, que ens permetran canviar les adreces de memòria dels registres de retorn de les funcions. Seguirem amb els conceptes de l'execució de processos, i això ens permetrà veure on i com s'emmagatzemen les variables, i ho aprofitarem per a donar algun exemple de variació dels valors.

Per acabar la primera part, veurem les eines més comunes que s'usen per a seguir el comportament d'un programa i poder canviar-lo. Aquests depuradors ens permetran entendre què fa el programa a cada moment i com el podem modificar perquè es comporti com nosaltres vulguem.

Una vegada tinguem clars els conceptes i les eines que hem d'utilitzar, descriurem els *exploits* de programació existents.

Veurem com es fa per a variar el comportament d'un programa i com podem generar un *exploit* de manera automàtica, usant eines ja existents que generen els programes que exploten una vulnerabilitat d'un determinat sistema operatiu.

A més de veure com es poden atacar els sistemes, veurem com es protegeixen les noves versions dels sistemes operatius d'aquests atacs clàssics. La conscienciació generalitzada dels desenvolupadors de programari que la mala programació genera enormes pèrdues de diners fa que cada vegada hi hagi menys vulnerabilitats i els sistemes operatius siguin cada vegada més segurs. Això ho demostra la poca quantitat de vulnerabilitats donades a conèixer en relació amb les últimes versions dels sistemes actuals comparades amb les anteriors.

2. Gestió de memòria

Per a crear o comprendre un *exploit* és molt important conèixer profundament l'arquitectura i el funcionament de la màquina que es vol atacar. Molts *exploits* aprofiten el funcionament normal d'un sistema per a obtenir-hi control, aprofitant errors que el programador de l'aplicació/sistema ha comès. Com a cas d'estudi ens centrarem en l'arquitectura dels processadors d'Intel. Altres processadors o arquitectures maquinari poden processar la informació de manera diferent. Així, doncs, un *exploit* és totalment dependent de l'arquitectura del sistema que es vol atacar.

Nota

Intel. Disponible a http://www.intel.com/

2.1. Ordre d'escriptura de bits i bytes en memòria

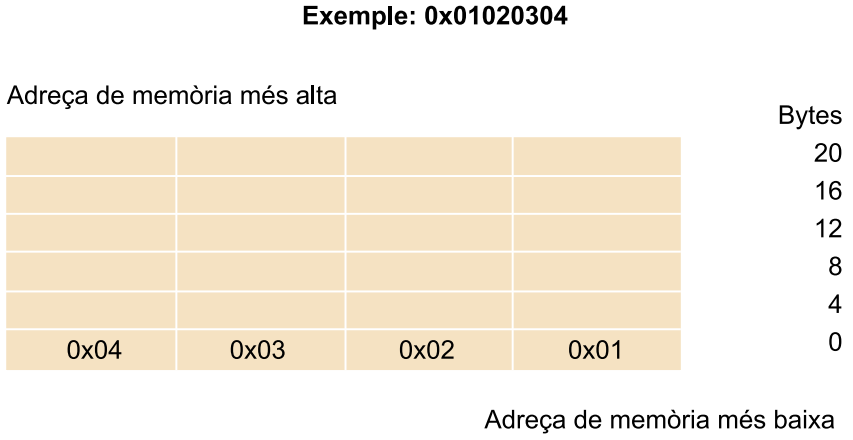
Hi ha diverses convencions per a escriure dades en memòria. Les més conegudes són les escriptures en *little-endian* i *big-endian*.

Little-endian escriu el byte menys significatiu (*least significant byte*, LSB) en l'adreça de memòria més baixa. En l'exemple següent, s'escriu en memòria una dada seguint aquest sistema.

Exemple: 0x01020304

Little-endian

Big-endian, al contrari de *little-endian*, escriu el byte més significatiu (*most significant byte*, MSB) en l'adreça de memòria més baixa.



Big-endian

Els sistemes basats en l'arquitectura Intel (x86 fins a IA-32 i Intel 64) funcionen usant el model *little-endian*.

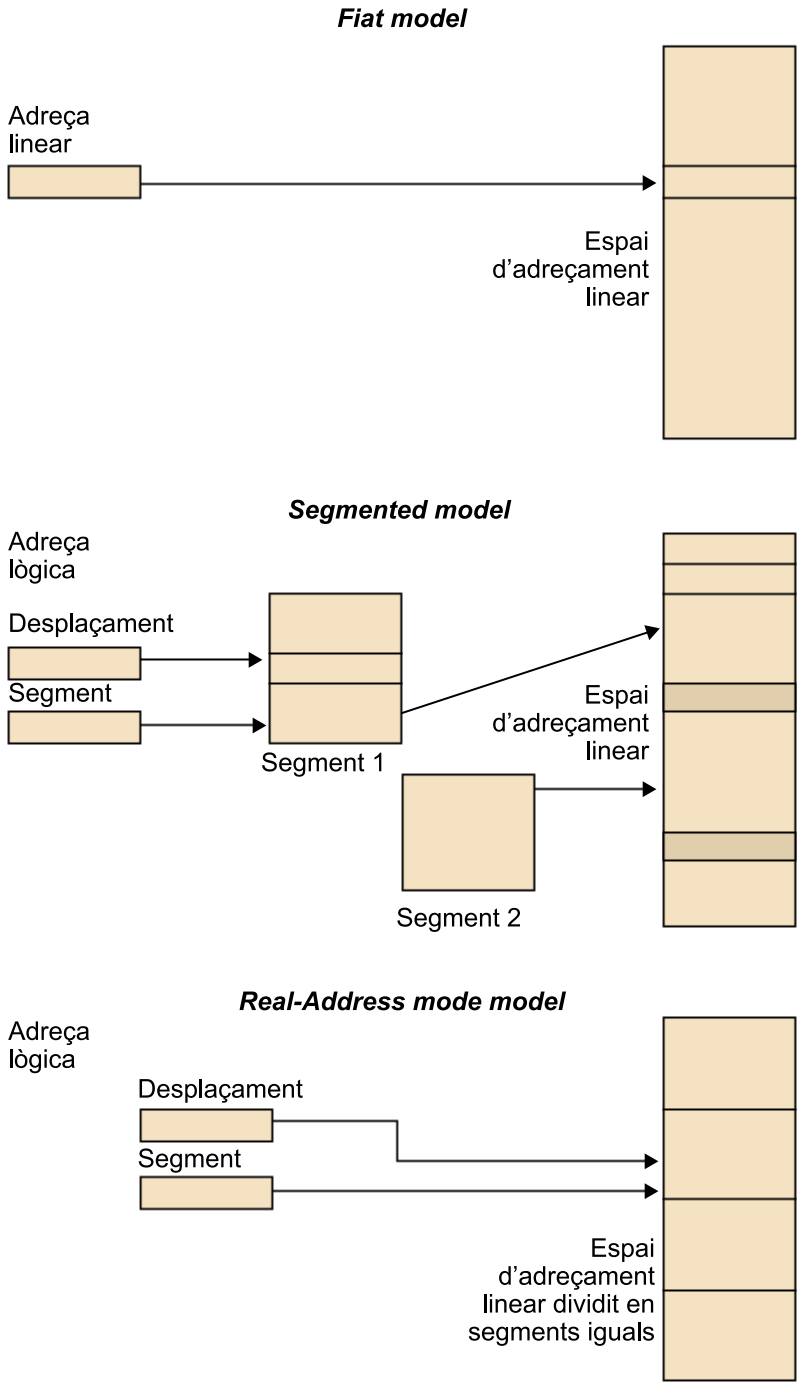
2.2. Segments

Quan un programa necessita accedir a memòria, no accedeix directament a la memòria física, sinó que usa un dels models de gestió de memòria que el processador posa al seu abast. En els sistemes basats en x86 trobem 3 models de memòria possibles:

- *Flat memory model*. Aquest model mostra la memòria al programa com un espai d'adreçament únic i contigu. Totes les dades necessàries per a l'execució del programa es troben en el mateix espai d'adreçament (dades, pila, programa).

- *Segmented memory model*. Aquest model és el més usat pels programes i mostra la memòria com un grup d'espais d'adreçament independents anomenats *segments*. Així, doncs, hi ha segments diferents per a usos diferents: dades, pila, codi. Per a accedir a un byte d'un segment els programes fan una petició d'una adreça lògica. Aquesta adreça conté el segment a què es vol accedir més un desplaçament dins d'aquest (òfset). Internament el processador fa la conversió de l'adreça lògica a l'adreça final dins de l'espai d'adreçament del processador.
  L'ús de segments augmenta la fiabilitat dels sistemes, i evita així que la pila creixi fins a sobreescriure el codi del programa, entre altres coses.

- *Real-address mode memory model*. Aquest model existeix per a oferir compatibilitat amb programes escrits per a processadors 8086, i en permet així l'ús en processadors més moderns. Amb els models de memòria *flat* o *segmented* l'espai d'adreçament linear s'usa per mitjà de l'espai d'adreçament físic del processador, directament o mitjançant paginació. Si la paginació no s'usa, cada adreça sol·licitada pel processador té correspondència directa amb una adreça física.



Models de gestió de memòria

Si s'usa paginació, l'espai d'adreçament linear es divideix en fragments anomenats *pàgines* i aquestes passen a formar part de la "memòria virtual". Aquestes pàgines de dades són gestionades de manera transparent pel sistema. Les aplicacions només veuen un espai d'adreçament linear al qual tenen accés, però aquest es gestiona d'una manera diferent del que veuen les aplicacions.

2.3. Registres bàsics per a l'execució de programes

Un dels elements més importants per a gestionar el sistema són els registres.

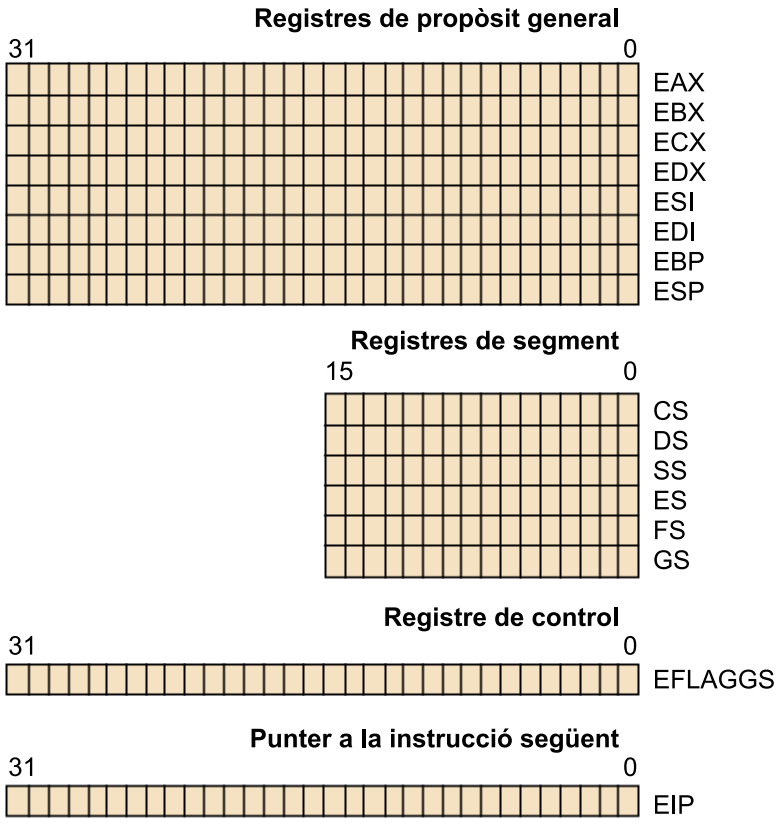
Aquests registres bàsics en una arquitectura x86 es poden agrupar en les categories següents:

- Registres de propòsit general. Són vuit registres de propòsit general per a desar dades i punters.

- Registres de segment. Hi ha fins a un total de 6 selectors de segments.

- Registre *EFLAGS*. És un registre especial encarregat de portar el control de certes operacions en l'execució d'un programa, i pot tenir certa incidència sobre el processador.

- Registre *EIP*. Aquest registre conté sempre un punter a la instrucció següent que ha d'executar el processador. Té especial interès quan s'està atacant una màquina mitjançant un *exploit*. L'objectiu principal és canviar aquest registre i assignar un punter a una adreça de memòria on hàgim pogut carregar el nostre codi.



Registres del sistema

2.3.1. Registres de propòsit general

Els registres de propòsit general són els següents: EAX, EBX, ECX, EDX, ESI, EDI, EBP i ESP. S'usen per a fer càlculs numèrics, càlculs d'adreces de memòria i també com a punters a memòria.

Encara que tots es poden usar d'una manera general, la majoria tenen usos predefinits. Per exemple, el registre ESP s'usa per a controlar la pila. Així, doncs, és altament recomanable (o obligatori) no usar-lo per a cap altra funció. Altres registres amb utilitats preestablertes són l'EBP (per al control dels contextos d'execució i accés als paràmetres en la pila), o els ECX, ESI i EDI, que s'utilitzen en les operacions amb cadenes de caràcters (*strings*).

Els usos especials de cada registre són els següents:

- EAX. També anomenat *acumulador*. Serveix com a recipient per a desar el resultat de moltes operacions. També s'usa per a retornar valors en crides a funcions.

- EBX. Punter a dades que usa el segment de dades (DS).

- ECX. També anomenat *comptador*. Se sol usar com a comptador en bucles.

- EDX. Punter d'entrada/sortida.

- ESI. Punter a dades que usa el segment de dades (DS). Punter a les dades font per a operacions relacionades amb *strings*.

- EDI. Punter a dades que usa el segment de dades (DS). Punter a l'adreça destinació per a operacions relacionades amb *strings*.

- ESP. Punter de pila (*stack pointer*). Controla el nivell de la pila.

- EBP. Punter a la pila (*base pointer*). Apunta a dades en la pila.

Tots aquests registres són de 32 bits, tal com es mostra en el gràfic anterior. Tot i així, per mantenir la compatibilitat amb sistemes antics, els mateixos registres existeixen en una arquitectura de 16 bits, però els seus noms canvien, perden la *E* que els precedeix. En la versió de 16 bits els noms d'aquests registres serien AX, BX, CX, DX, SI, DI, SP, BP. Encara que el nom canvia lleugerament, les funcions assignades continuen essent les mateixes. A més d'aquesta nomenclatura, es pot accedir separadament al byte alt i baix dels registres AX, BX, CX i DX canviant la *X* per la *H* de *high* ('part alta') o la *L* de *low* ('part baixa').

- Registres de 32 bits (*doubleword registers*): EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP.
- Registres de 16 bits (*word registers*): AX, BX, CX, DX, SI, DI, SP, BP.
- Accés a bytes dels registres AX, BX, CX i DX (*byte registers*): AH, AL, BH, BL, CH, CL, DH i DL.

2.3.2. Registres de segments

Els registres de segments són sis. La mida de tots és de 16 bits i s'usen per a crear segments en memòria que seran destinats a usos diferents. Aquests registres són CS (*code segment*), DS (*data segment*), SS (*stack segment*), ES (*extra segment*), FS i GS.

Els quatre segments de memòria principals (CS, DS, SS i ES) ja tenen un ús establert.

- CS. Segment en el qual s'emmagatzema el programa en execució. El processador obté les instruccions per executar utilitzant aquest segment i el registre EIP. El registre EIP conté el desplaçament des del principi de segment definit per CS, fins a la instrucció següent que s'ha d'executar. El registre CS no pot ser canviat de manera explícita per un programa. Aquest registre és gestionat de manera implícita pel sistema.

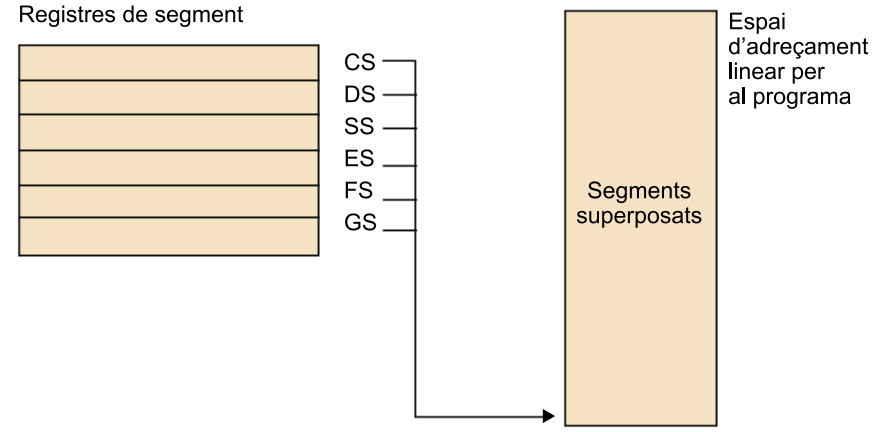
- DS. S'hi desen les dades que utilitza el programa.

- SS. En aquest segment es crea la pila de dades, necessària per a l'execució de tots els programes. Totes les operacions que necessiten la pila usen aquest segment per a accedir-hi. Aquest registre sí que pot ser canviat per un programa. Això permet la creació de diverses piles per a un programa i poder canviar entre aquestes segons que es requereixi.

- ES. Segment de dades extra per a poder emmagatzemar més dades en memòria.

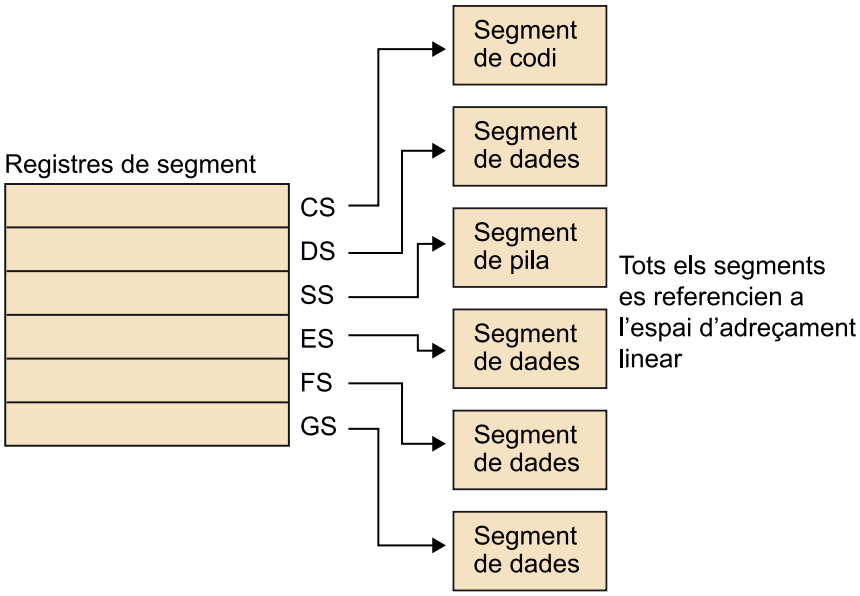
Els segments FS i GS es van introduir més tard en l'arquitectura x86 i s'utilitzen com a segments de dades addicionals igual que el segment ES.

La utilització dels segments depèn del model de memòria utilitzat. Si s'utilitza el model de memòria *flat*, llavors tots els segments apuntaran al mateix segment de memòria, i en resultarà un sistema amb tots els segments superposats.



Registres de segment que usen el model *flat*

Si s'utilitza un model de memòria segmentat, llavors es poden crear segments diferents que apuntin a segments de memòria totalment separats. D'aquesta manera un programa pot accedir a sis segments diferents en l'espai de memòria. En el cas de voler accedir a un segment del qual no disposa, primer ha de carregar el valor apropiat en el registre de segment i després accedir a les dades del nou segment.



Registres de segment que usen el model *segmented*

2.3.3. Registre EFLAGS

El registre EFLAGS porta el control d'una sèrie d'indicadors del sistema, entre els quals hi ha indicadors d'estat, de control i de sistema. Cada bit d'aquest registre indica una situació concreta. Durant l'execució d'un programa aquest registre va modificant els seus bits depenent dels resultats de les diferents operacions que s'executen en el programa. Aquests bits poden ser llegits i interpretats per a usar-los durant l'execució del programa. Hi ha diverses instrucci-

ons que permeten moure el contingut d'aquest registre a la pila o al registre EAX. Una vegada es té el valor en un d'aquests dos llocs se'n pot inspeccionar el contingut. No es pot canviar el valor del registre EFLAGS de manera directa.

Els diferents bits que es troben en aquest registre se solen anomenar *indicadors* i els que no estan reservats (no es poden usar) són els següents:

- *ID Flag* (ID). Indicador de sistema.
- *Virtual interrupt pending* (VIP). Indicador de sistema.
- *Virtual interrupt flag* (VIF). Indicador de sistema.
- *Alignment check* (AC). Indicador de sistema.
- *Virtual-8086 code* (VM). Indicador de sistema.
- *Resume flag* (RF). Indicador de sistema.
- *Nested task* (NT). Indicador de sistema.
- *I/O privilege level* (IOPL). Indicador de sistema.
- *Overflow flag* (OF). Indicador d'estat.
- *Direction flag* (DF). Indicador de control.
- *Interrupt enable flag* (IF). Indicador de sistema.
- *Trap flag* (TF). Indicador de sistema.
- *Sign flag* (SF). Indicador d'estat.
- *Zero flag* (ZF). Indicador d'estat.
- *Auxiliary carry flag* (AF). Indicador d'estat.
- *Parity flag* (PF). Indicador d'estat.
- *Carry flag* (CF). Indicador d'estat.

Indicadors d'estat

Aquests indicadors donen indicacions sobre el resultat d'operacions aritmètiques com ara les instruccions ADD o SUB. Els usos dels diferents bits d'estat són els següents:

- *Carry flag* (CF). Indica si l'operació ha generat un resultat més gran que el registre resultat és capaç de contenir.

- *Parity flag* (PF). Indica si el bit de menys pes del resultat és un 1. Detecció de resultats imparells.

- *Adjust flag* (AF). Indica si l'operació ha generat un resultat més gran que el bit 3 del resultat.

- *Zero flag* (FG). Indica si el resultat és zero.

- *Sign flag* (SF). Adquireix el valor del bit de més pes del resultat de l'operació. En operacions amb enters 0 indica un nombre positiu i 1 indica un nombre negatiu.

- *Overflow flag* (OF). En operacions amb enters aquest bit indica si el resultat és massa gran (nombres positius) o massa petit (nombres negatius) per a cabre en el contenidor de destinació.

Indicadors de control

L'únic indicador en aquesta categoria és el *direction flag* (DF). Aquest indicador indica en quina adreça es processaran les instruccions relacionades amb *strings*. Això significa si les operacions amb *strings* seran tractades amb autoincrements (bit a 0, d'adreces de memòria baixes a adreces altes), o amb autodecrements (bit a 1, d'adreces de memòria altes a adreces baixes). Hi ha dues instruccions que canvien aquest indicador: STD (bit a 1), CLD (bit a 0).

Indicadors de sistema

Aquests indicadors controlen el comportament del sistema, de manera que no han de ser canviats per les aplicacions. Els diferents indicadors de sistema són els següents:

- *Trap flag* (TF). A 1, activa el traçat (*debug*) pas per pas.

- *Interrupt enable flag* (IF). A 1, activa la resposta a les interrupcions. A 0, desactiva les interrupcions.

- *I/O privilege level field* (IOPL). Aquest camp està compost per dos bits i indica el privilegi actual del programa en curs per a accedir a l'espai d'entrada/sortida.

- *Nested taks flag* (NF). Controla l'encadenament de tasques en interrupcions o crides.

- *Resume flag* (RF). Controla la resposta del processador a les excepcions de traçat (*debug*).

- *Virtual-8086 mode flag* (VM). A 1 per a activar el mode 8086.

- *Alignment check flag* (AF). Controla si s'ha de fer prova d'alineació de referències a memòria.

- *Virtual interrupt flag* (VIF). Usat en el control de les interrupcions.

- *Virtual interrupt pending flag* (VIP). A 1 indica que hi ha una interrupció pendent.

- *Identification flag* (ID). Dóna suport per a l'ordre CPUID.

Registre EIP

Aquest registre controla l'execució del programa. Conté un desplaçament en memòria, comptat tenint en compte el segment de codi (CS) que indica quina serà la propera instrucció que s'ha d'executar. Aquest registre no pot ser alterat directament, però hi ha diverses operacions que hi tenen impacte: JMP, Jcc, CALL, RET i IRET (les veurem més endavant).

Una manera d'accedir al valor del registre EIP és executar una instrucció CALL (la qual escriu en la pila el valor d'EIP, entre altres coses), i després llegir el valor d'EIP en la pila mateixa. Tot i així, serà el valor de retorn de la instrucció CALL, i mai el valor actual del registre EIP. Estudiarem aquest procés detalladament més endavant, ja que els *exploits* fan servir aquests registres per a saltar de funció.

2.4. Utilització de la pila

La pila és un dels recursos importants en el funcionament dels sistemes. És un recurs que ha de ser administrat amb summa cura ja que en depèn, en part, el bon funcionament del sistema.

La pila és una manera de desar dades que es basa en una estructura LIFO (*last in, first out*). Així, doncs, els últims elements a arribar a la pila són els primers que sortiran. Aquesta manera de procedir encaixa perfectament amb el comportament de les estructures d'execució dels programes.

Per això la pila s'usa per a portar el control de les crides a funcions i procediments de manera imbricada en els programes.

Com ja hem comentat, en l'arquitectura x86 s'usen els registres SS (*stack segment*) i SP (*stack pointer*) com a registres bàsics per a accedir a la pila. A aquests s'ha d'afegir el registre BP (*base pointer*), que s'usa per a accedir a dades l'emplaçament de les quals és en la pila. BP usa el segment SS per defecte, igual que el registre SP. Aquest últim registre s'encarrega de controlar la part superior de la pila (*top*).

Informacions que es desen en la pila:

- Adreça de retorn. Una de les dades més importants que es desen en la pila és l'adreça de retorn de l'execució del programa. Quan es crida una funció o procediment, l'adreça de retorn de l'execució del programa es desa en la pila. Després d'executar la funció o procediment, es reprèn l'execució en el punt on s'havia interromput.

- Espai per a dades locals. Les variables declarades de manera local a una funció són també desades en la pila. Així, doncs, desapareixen totalment després de l'execució.

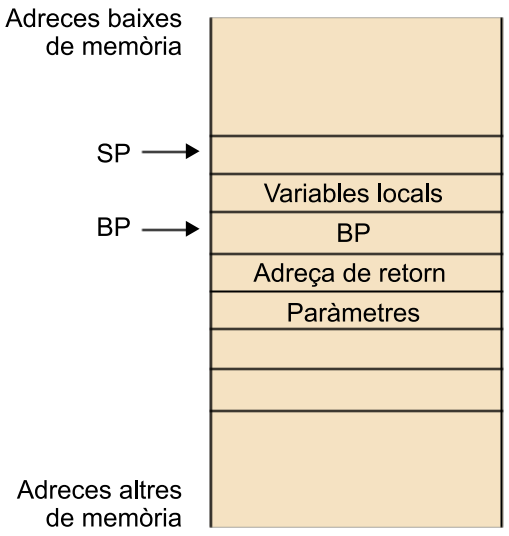
- Pas de paràmetres. Els paràmetres necessaris per a l'execució de les funcions cridades també es desen en la pila.

- Retorn de resultats. La pila també s'usa per a poder retornar els resultats de les funcions cridades.

- Estats de l'execució. En determinades situacions també es desen en la pila dades corresponents a l'estat de l'execució, com per exemple el registre EFLAGS.

- Espai per a càlculs. De vegades s'utilitza aquest recurs per a fer càlculs amb diferents operands.

L'esquema de l'organització de la pila durant l'execució d'un programa és el següent:

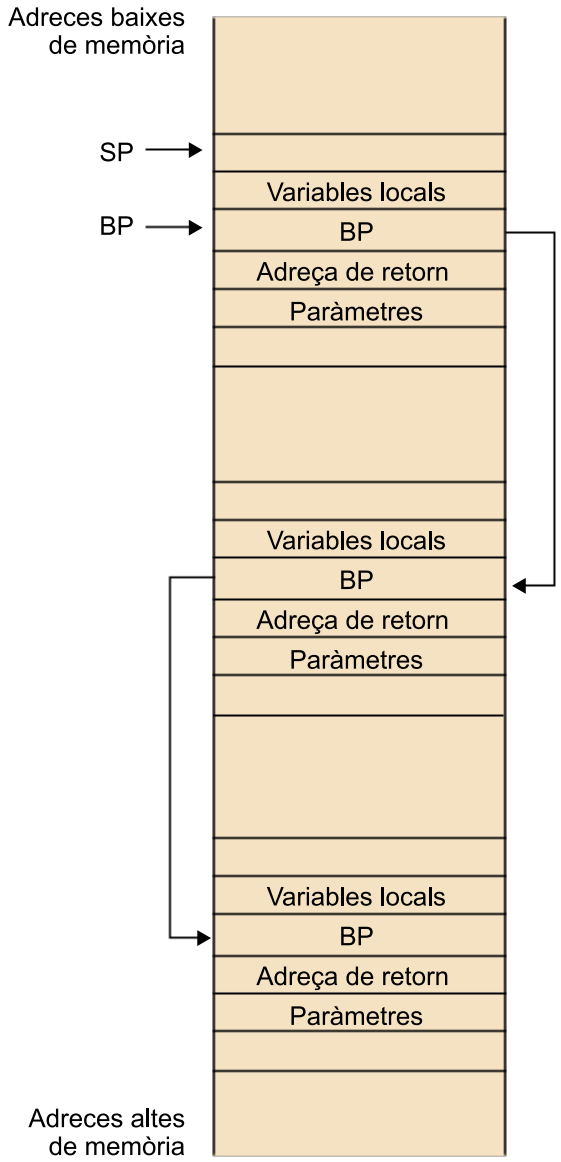


Estructura del contingut de la pila

En el gràfic anterior veiem l'estructura del contingut de la pila, més els dos registres que s'usen per a accedir a les dades que s'hi troben, juntament i de manera transparent amb el registre de segment SS. El registre SP sempre apunta a la part superior de la pila (*top*), i el registre BP apunta a l'últim valor desat, que al seu torn apunta al seu valor desat anterior en la pila. Això produeix una sèrie d'encadenaments que serveix per a mantenir el seguiment dels diferents contextos d'execució de les diferents funcions cridades.

També s'aprecia que la pila comença en les adreces altes de memòria i va creixent cap a les adreces baixes. Cada vegada que s'introdueix un element en la pila, el registre SP es disminueix, i augmenta així la mida de la pila. Per a alliberar elements de la pila el registre SP s'incrementa, i es redueix així la mida de la pila.

Si s'observa com queda la pila després de fer crides successives a funcions o procediments, s'obté una estructura com la següent:



Encadenament de les estructures de dades de la pila

Així com el registre SP és un valor volàtil (sempre apunta a la part superior de la pila, que pot ser molt variable), el contingut del registre BP és bastant estable. És sempre un apuntador al seu valor antic en la pila, per la qual cosa és ideal utilitzar-lo per a accedir a valors que s'hi desen. Aquest registre s'usa sobretot per a accedir als paràmetres que es passen a les funcions i a les variables locals a les quals s'ha reservat espai en la pila.

3. Conceptes bàsics de llenguatge màquina

Tot programa fet en un llenguatge d'alt nivell és transformat perquè pugui ser executat per la màquina per a la qual s'ha fet. En aquest procés es fan dos passos importants: compilació i enllaçament (*link*). Un cas a part seria la utilització de llenguatges interpretats.

El resultat de fer aquests dos processos és un programa executable que està escrit en un llenguatge que la màquina és capaç d'executar (codi màquina). Durant la fase de compilació es fa la traducció del llenguatge font a codi màquina i durant la fase d'enllaçament (*link*) s'enllaça aquest codi final amb totes les biblioteques necessàries perquè el programa pugui ser executat. Aquest enllaçament pot ser estàtic (totes les biblioteques necessàries es passen a incloure en el resultat final) o dinàmic (el codi final queda preparat per a enllaçar-se a les biblioteques en el moment de l'execució).

Com que és el codi màquina el que s'acaba executant, és important conèixer uns quants detalls sobre com funciona, més concretament el llenguatge assemblador.

El codi màquina és el llenguatge que l'ordinador és capaç d'executar directament, però és difícil de llegir per a les persones. Són només codis en binari (per a comprendre'ls millor se solen llegir en hexadecimal) que indiquen al processador què és el que ha d'anar fent. A causa d'aquesta dificultat es van crear una sèrie d'etiquetes que descriuen aquests codis per fer-los així més intel·ligibles. Aquesta sèrie d'etiquetes és el que es denomina *llenguatge assemblador*, i és el llenguatge de programació més proper al llenguatge màquina. El llenguatge assemblador també ha de ser compilat, però la traducció a llenguatge màquina sol ser directa. Així, doncs, és possible seguir l'execució d'un programa en memòria usant eines de traçat (depuradors) en el nivell de llenguatge assemblador.

Cada processador té un conjunt d'instruccions que és capaç d'executar. Aquestes instruccions s'executen en el nivell de registre o memòria. Així, doncs, hi ha diversos tipus d'operacions que es poden fer:

- Instruccions de transferència de dades: MOV, PUSH, POP...
- Operacions binàries entre enters: ADD, SUB, MUL, DIV...
- Aritmètica decimal: DAA, DAS...
- Operacions lògiques: AND, XOR, OR, NOT...
- Desplaçaments i rotacions: SHR, SHL, ROR, ROL...
- Operacions de bits i bytes: BTS, BTR, BSR, TEST...
- Operacions per al control del programa: JMP, JE, JZ, CALL, RET...
- Operacions per a cadenes de caràcters: MOVS, CMPS, REP...

- Operacions d'entrada/sortida: IN, OUT, INS...
- Control d'indicadors: STC, CLC, PUSHF, POPF...
- Operacions de registres de segments: LDS, LES, LFS...
- Diverses de no incloses en les categories anteriors: LEA, NOP, CPUID...

Depenent del tipus de processador, hi haurà altres subconjunts d'operacions disponibles. És tasca del compilador usar un determinat subconjunt d'instruccions per a tenir així un rendiment òptim del programa final, si té dades sobre l'arquitectura final sobre la qual s'executarà el programa. Possibles subconjunts d'operacions disponibles:

- x87 FPU. Instruccions per a fer càlculs amb nombres en coma flotant.
- MMX. Extensions multimèdia.
- SSE. Extensions per a *streaming* SIMD.
- SSE-2. Ampliació de les extensions SSE per a poder fer operacions en coma flotant amb operands de 64 bits.
- SSE-3. Millora del rendiment de les extensions SSE.
- SSSE-3. Millora de rendiment amb enters.
- SSE-4, SSE-4.1, SSE-4.2. Subconjunts d'operacions pensats per a millorar el rendiment en determinats tipus d'aplicacions gràfiques.

3.1. Tipus de nomenclatures

Hi ha dos tipus de nomenclatures àmpliament utilitzades quan es treballa en assemblador. Són les següents:

- Intel
- AT&T[1]

La sintaxi d'Intel és la més utilitzada en entorns Windows[2] i la d'AT&T és molt utilitzada en entorns Unix[3]/Linux[4]. Quan s'està traçant un programa, primer cal veure quin tipus de nomenclatura usa l'eina.

Les dues nomenclatures tenen diferències:

- En la sintaxi d'Intel primer s'escriu l'operand de destinació i després el de font. Això és totalment a l'inrevés en la nomenclatura AT&T.
- En la sintaxi AT&T els codis d'operació/instruccions (*opcodes*) van seguits d'una lletra que indica la mida dels operands (*l* per a doble *word*, *w* per a *word* i *b* per a byte).
- En la sintaxi AT&T els valors van precedits d'un $ i els registres d'un %.

(1)**AT&T**. Disponible a http://www.att.com/

(2)**Microsoft Windows**. Disponible a http://www.microsoft.com/windows/

(3)**Unix**. Disponible a http://www.unix.org/

(4)**Linux**. Disponible a http://www.kernel.org/

- En la sintaxi AT&T les referències a posicions a memòria s'escriuen entre parèntesis, i en canvi en la sintaxi d'Intel s'escriuen entre claudàtors.

3.2. Operacions i operands en assemblador

Les instruccions en assemblador solen ser sentències curtes amb dos, un o cap operand explícit. Moltes de les ordres impliquen la utilització de registres de manera implícita.

Les operacions i operands en llenguatge màquina se solen llegir en hexadecimal, i com ja hem comentat, tenen una traducció directa en llenguatge assemblador. La sentència següent està escrita en llenguatge assemblador i executaria una assignació d'un valor (`0x110`) sobre el registre *edx*:

```
mov $0x110, %edx
```

Com es pot observar, la instrucció està escrita en sintaxi AT&T. Això indica que l'operand destinació és el segon i que el font és el primer. Aquesta sentència en llenguatge màquina (hexadecimal) seria:

```
ba 10 01 00 00
```

El primer que s'observa és com la màquina ha desat en memòria el valor `0x110` (si apliquem zeros a l'esquerra tenim el valor `0x00000110`). S'observa l'aplicació de la norma *little-endian* per a l'emmagatzematge de dades.

El primer byte indica el tipus d'operació *mov* i quin és el registre destinació *edx*. Els quatre últims bytes indiquen el valor que s'ha d'assignar al registre seleccionat. Passant el primer byte a binari s'observa el següent:

```
1011 1010
```

Els bits 4, 5, 6 i 7 (dada `1011`) indiquen que es tracta de l'operació *mov*, el bit 3 indica que es tracta de moure una dada de mida *double word* i els bits 0, 1 i 2 (dada `010`) indiquen que les dades s'han de desar en el registre *edx*.

El registre destinació estaria especificat per la relació següent:

- `000` → `eax`
- `001` → `ecx`
- `010` → `edx`
- `011` → `ebx`
- `100` → `esp`
- `101` → `ebp`
- `110` → `esi`
- `111` → `edi`

Tot i així, aquesta relació depèn del tipus d'instrucció que s'estigui executant. Si en lloc de ser una instrucció *mov* de mida *double word* hagués estat una instrucció de mida de byte, la relació que definiria el registre destinació seria una altra:

- `000 → al`
- `001 → cl`
- `010 → dl`
- `011 → bl`
- `100 → ah`
- `101 → ch`
- `110 → dh`
- `111 → bh`

Si es volgués fer un canvi en la mida dels operands i passar a una assignació de byte, la sentència en assemblador, encara que incorrecta, seria la següent:

```
mov $0x110, %dl
```

En aquesta sentència s'està intentant assignar un valor que supera el valor de la mida màxima d'un byte. El valor `0x110` és més gran que un byte, de manera que si es vol que la sentència sigui correcta, s'ha de canviar el valor o canviar el registre destinació.

```
mov $0x10, %dl
```

Aquesta sentència sí que és correcta i la seva traducció a codi màquina en hexadecimal és:

```
b2 10
```

S'observa que el codi generat és més curt que en la instrucció anterior, ja que la mida dels operands ha variat. Això té fortes implicacions si es modifica codi màquina en execució, ja que si es fa aquesta modificació de la primera sentència que s'ha estudiat:

```
ba 10 01 00 00      mov $0x110, %edx
b2 10               mov $0x10, %dl
```

s'aprecia que l'única cosa que ha canviat és el codi d'instrucció (ha passat de `ba` a `b2`). D'aquesta manera, la mida dels operands ha canviat i tres dels bytes que abans formaven part del valor per assignar al registre destinació queden ara sense funció.

Aquests bytes no queden sense fer res durant l'execució del programa, sinó que el sistema els interpreta com si fossin una nova instrucció. La norma que segueix el sistema és "on acaba una instrucció, en comença una altra". D'aquesta

manera, uns bytes que havien de ser interpretats com a tals passen a ser codi executable, i alteren així l'execució normal del programa. En l'exemple, els dos bytes següents serien executats com a:

```
01 00 add %eax,(%eax)
```

i a partir d'aquest punt s'executarien una infinitat de canvis a causa de la reinterpretació del codi màquina del sistema.

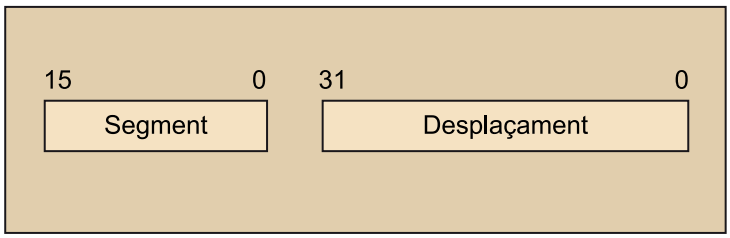
3.3. Adreçament i operands

Com hem vist, les instruccions en assemblador requereixen operands. El nombre d'operands pot variar de 0 (operands implícits a la instrucció) a diversos (operands explícits). Les dades que proveeixen els operands poden estar especificades pel següent:

- **Un valor explícit expressat en la instrucció (valor immediat)**. Els valors immediats tan sols poden ser usats com a operands font d'una instrucció, ja que estan escrits directament en la instrucció i, per tant, no expressen cap contenidor vàlid per a registrar dades. Són, solament, l'expressió d'un valor.
  Moltes operacions, sobretot les aritmètiques, accepten valors immediats com a operands font. El valor màxim expressat per aquests operands i acceptat per les diferents instruccions varia d'instrucció a instrucció.

- **Un registre**. Els registres poden ser especificats tant com a operands font o com a operands destinació. Tots els registres de propòsit general vistos poden ser usats (els de 32, 16 i 8 bits). Els registres de segment també. El registre EFLAGS és usat de manera implícita en algunes instruccions (com, per exemple, en les instruccions de salt condicional).

- **Una adreça de memòria**. Un operand que especifica una adreça de memòria sempre implica l'ús d'un segment i un desplaçament dins del segment.



Accés a memòria

La selecció del segment per utilitzar per a l'accés a memòria es pot fer de manera implícita o explícita. El més habitual és carregar els diferents registres de segment amb els valors corresponents i deixar que la instrucció executada seleccioni el segment per defecte. El processador segueix una sèrie de regles per a escollir els registres de segment durant l'execució d'una instrucció:

- El registre de segment CS s'usa sempre que cal accedir a una instrucció nova per a carregar-la al registre EIP.

- El registre de segment SS s'usa per als accessos a la pila. Totes les instruccions que utilitzin els registres EBP o ESP per a accedir a dades de la pila usaran el registre SS com a registre de segment base.

- El registre de segment DS s'usa quan s'accedeix a referències a dades, excepte aquelles que se situïn en la pila, o facin referència a la destinació de cadenes de caràcters (*strings*).

- El registre de segment ES per a situar la destinació de les instruccions que facin operacions sobre cadenes de caràcters.

Encara que els segments s'usen moltes vegades de manera implícita, també es poden usar de manera explícita, expressant el segment que es vol usar per a alterar així l'ordre preestablert pel sistema.

```
mov $0x15, %es:%ebx
```

Amb la instrucció anterior es carregaria el valor 15 en hexadecimal, en l'adreça de memòria expressada per la combinació del registre ES (segment) i EBX (desplaçament).

Encara que és possible fer aquest tipus de crides, hi ha algunes normes que no poden ser alterades.

- Les instruccions sempre es carregaran usant el registre CS.
- La destinació de les operacions de *strings* sempre serà situada en el segment ES.
- Les instruccions de *push* i *pop* sempre s'executen usant el registre SS.

L'accés a memòria és un punt clau en l'arquitectura de computadors. Per a accedir-hi hi ha diversos mètodes d'adreçament en els quals els desplaçaments s'expressen de diferents maneres per a poder accedir a les dades. Aquest desplaçament es pot expressar com un valor estàtic o com el resultat d'un o diversos components:

- Punter base.
- Índex.
- Escala.

- Desplaçament.

Aquests quatre elements se solen combinar mitjançant la fórmula següent:

punter_base + índex * escala + desplaçament

Aquest càlcul és especialment útil per a accedir a matrius de dades desades en memòria. No cal usar tots els elements en cada accés a memòria. A continuació esposarem alguns exemples d'accés a memòria en la sintaxi d'Intel i la d'AT&T:

- Accés a les dades apuntades per l'adreça de memòria continguda en un registre:

AT&T:	`(%eax)`
Intel:	`[eax]`

- Afegim un desplaçament:

AT&T:	`_despl(%eax)`
Intel:	`[eax + _despl]`

- Accés a un valor en una matriu d'enters de mida 4 bytes:

AT&T:	`_array(,%eax,4)`
Intel:	`[eax*4 + _array]`

- Accés a un caràcter en una matriu de registres de 8 caràcters, en què *eax* conté el nombre de registres desitjat i *ebx* té el desplaçament desitjat dins del registre:

AT&T:	`_array(%ebx, %eax, 8)`
Intel:	`[ebx + eax*8 + _array]`

A escala de codi màquina les diferents combinacions de desplaçament, base, escala i índex es codifiquen en una instrucció. Els compiladors de llenguatges d'alt nivell seleccionen els elements més apropiats per a accedir a memòria tenint en compte les estructures creades pels programadors.

- **Un port d'entrada/sortida**. A l'espai d'entrada/sortida s'hi accedeix mitjançant un conjunt de ports. El nombre de ports disponible és limitat. Aquests són accessibles mitjançant un operand amb un valor immediat o bé un valor en el registre DX.

3.4. Desassemblament d'un petit programa

Vista la petita introducció al llenguatge assemblador, es considera el programa següent en llenguatge C.

```
#include <stdio.h>
int main()
{

   int x, y, result;

   x = 3;

   y = 4;

   result = x + y;

   printf("Resultat de x + y = %d\n", result);

   return(1);
}
```

El programa tan sols fa la suma de dues variables locals (`x` i `y`) declarades de manera local en el cos principal del programa. El resultat és desat en la variable `result` per a ser posteriorment mostrat per pantalla mitjançant la funció `printf`.

Es procedeix a compilar el programa i posteriorment a desassemblar, per a veure quin és el resultat en nivells de programació més baixos: codi màquina i assemblador. Es fa l'operació en dos entorns completament diferents: Linux + GCC[5] i Windows + BCC[6].

(5)**GCC, the GNU Compiler Collection**. Disponible a http://gcc.gnu.org/

(6)**Borland C++ compiler**. Disponible a http://www.codegear.com/downloads/free/cppbuilder

3.4.1. Entorn: Linux + GCC

El codi resultant de desassemblar l'executable, resultat de compilar el programa anterior centrant l'objectiu en el codi escrit en la funció *main*, és:

```
Columna 1                        Columna 2     Columna 3

080483a4 <main>:
080483a4: 8d 4c 24 04               lea      0x4(%esp),%ecx
080483a8: 83 e4 f0                  and      $0xfffffff0,%esp
080483ab: ff 71 fc                  pushl    -0x4(%ecx)
080483ae: 55                        push     %ebp
080483af: 89                        e5 mov   %esp,%ebp
080483b1: 51                        push     %ecx
080483b2: 83 ec 24                  sub      $0x24,%esp
080483b5: c7 45 f0 03 00 00 00      movl     $0x3,-0x10(%ebp)
080483bc: c7 45 f4 04 00 00 00      movl     $0x4,-0xc(%ebp)
080483c3: 8b 55 f4                  mov      -0xc(%ebp),%edx
080483c6: 8b 45 f0                  mov      -0x10(%ebp),%eax
080483c9: 01 d0                     add      %edx,%eax
080483cb: 89 45 f8                  mov      %eax,-0x8(%ebp)
080483ce: 8b 45 f8                  mov      -0x8(%ebp),%eax
080483d1: 89 44 24 04               mov      %eax,0x4(%esp)
080483d5: c7 04 24 b0 84 04 08      movl     $0x80484b0,(%esp)
080483dc: e8 f7 fe ff ff            call     80482d8 <printf@plt>
080483e1: b8 01 00 00 00            mov      $0x1,%eax
080483e6: 83 c4 24                  add      $0x24,%esp
080483e9: 59                        pop      %ecx
080483ea: 5d                        pop      %ebp
080483eb: 8d 61 fc                  lea      -0x4(%ecx),%esp
080483ee: c3                        ret
080483ef: 90                        nop
```

La sintaxi AT&T està molt estesa en el món Unix, de manera que la majoria d'utilitats que treballen amb aquestes plataformes la sol usar, encara que hi ha eines que també permeten el desassemblatge en la sintaxi d'Intel.

En la llista resultant hi ha poques coses identificables a primera vista. Les diferents columnes indiquen el següent:

- Primera columna. Adreça de memòria on es troba la instrucció. Aquestes són les adreces de memòria que s'aniran carregant en el registre EIP per a seguir l'execució, instrucció per instrucció, del programa.

- Segona columna. Es mostren els codis en llenguatge màquina *(opcodes)* de cada instrucció, resultat de la compilació del programa.

- Tercera columna. Instruccions en llenguatge assemblador resultat de desassemblar els codis en llenguatge màquina de la segona columna.

Així, doncs, en cada línia s'observa l'adreça de memòria de la instrucció, els bytes corresponents al codi màquina i la instrucció corresponent al codi màquina en llenguatge assemblador.

Si s'observa la línia:

```
080483dc: e8 f7 fe ff ff   call 80482d8 <printf@plt>
```

es pot veure com el sistema fa una crida a una funció anomenada `printf`. Es produeix, en aquest punt, un salt en l'execució lineal del programa a l'adreça `080482d8`, que és on es troba situada la funció `printf`.

El codi màquina indica la funció `call` mitjançant l'*opcode* `E8`. Els 4 bytes restants indiquen el desplaçament. Si es té en compte que les dades s'emmagatzemen seguint la norma *little endian* es veu que el desplaçament indicat és `fffffef7`, que passat a decimal es correspon amb el nombre –265.

Si es fa la resta entre l'adreça destinació i l'adreça a la instrucció següent a l'execució del *call*, s'obté que:

```
Hexadecimal: 80482d8 - 80483e1 = fffffef7
Decimal: 134513368 - 134513633 = -265
```

Obtenim així el desplaçament indicat en el codi màquina. Això indica que la crida serà un `call near` a una adreça dins del mateix segment de codi. És important recordar això, ja que els *exploits* modifiquen l'adreça de retorn.

A continuació s'intercala el codi font escrit en C amb el codi en assemblador (més endavant parlarem d'una eina que ens permetrà veure aquest codi assemblador a partir de codi C):

```
Adreça                          Opcodes         Assembler

080483a4 <main>:
#include <stdio.h>

int main()
{
80483a4: 8d 4c 24 04               lea          0x4(%esp),%ecx
80483a8: 83 e4 f0                  and          $0xfffffff0,%esp
80483ab: ff 71 fc                  pushl        -0x4(%ecx)
80483ae: 55                        push         %ebp
80483af: 89 e5                     mov          %esp,%ebp
80483b1: 51                        push         %ecx
80483b2: 83 ec 24                  sub          $0x24,%esp
int x, y, result;

x = 3;
80483b5: c7 45 f0 03 00 00 00      movl         $0x3,-0x10(%ebp)
y = 4;
80483bc: c7 45 f4 04 00 00 00      movl         $0x4,-0xc(%ebp)

result = x + y;
80483c3: 8b 55 f4                  mov          -0xc(%ebp),%edx
80483c6: 8b 45 f0                  mov          -0x10(%ebp),%eax
80483c9: 01 d0                     add          %edx,%eax
80483cb: 89 45 f8                  mov          %eax,-0x8(%ebp)

printf("Resultat de x + y = %d\n", result);
80483ce: 8b 45 f8                  mov          -0x8(%ebp),%eax
80483d1: 89 44 24 04               mov          %eax,0x4(%esp)
80483d5: c7 04 24 b0 84 04 08      movl         $0x80484b0,(%esp)
80483dc: e8 f7 fe ff ff            call         80482d8 <printf@plt>

return(1);
80483e1: b8 01 00 00 00            mov          $0x1,%eax
80483e6: 83 c4 24                  add          $0x24,%esp
80483e9: 59                        pop          %ecx
80483ea: 5d                        pop          %ebp
80483eb: 8d 61 fc                  lea          -0x4(%ecx),%esp
80483ee: c3                        ret
}
```

Es poden veure les línies de codi en C intercalades enmig de les línies de codi en assemblador. Les línies en assemblador que apareixen després de cada línia de codi en C són les necessàries perquè la instrucció en C sigui executada. Hi ha una excepció a aquesta norma, i és la declaració de variables a l'entrada de la funció `main`. La declaració dels enters apareix sense línies en assemblador per executar-se. Això no significa que aquesta línia no tingui repercussió en el nivell d'assemblador, sinó que les operacions necessàries per a l'execució d'aquesta instrucció ja s'han executat prèviament.

```
int x, y, result;
```

Com que són variables locals, la traducció de la declaració de les variables seria la reserva d'espai en la pila per a aquestes últimes. Aquesta reserva d'espai es fa en la línia immediatament anterior a la declaració de les variables:

```
080483b2: 83 ec 24    sub $0x24,%esp
```

Aquesta instrucció reserva 36 bytes en la pila per a ser utilitzats de manera local. Entre els usos es troba la utilització de les variables locals.

3.4.2. Entorn: Windows + BorlandC (BCC)

El codi resultant de desassemblar l'executable, resultat de compilar el programa anterior centrant l'objectiu en el codi escrit en la funció `main`, és:

```
Columna 1      Columna 2      Columna 3

00401150 >/.   55             PUSH EBP
00401151 |.    8BEC           MOV EBP,ESP
00401153 |.    B8 03000000    MOV EAX,3
00401158 |.    BA 04000000    MOV EDX,4
0040115D |.    03D0           ADD EDX,EAX
0040115F |.    8BC2           MOV EAX,EDX
00401161 |.    50             PUSH EAX
00401162 |.    68 28A14000    PUSH EXEMPLE1.0040A128
00401167 |.    E8 10270000    CALL EXEMPLE1.0040387C
0040116C |.    83C4 08        ADD ESP,8
0040116F |.    B8 01000000    MOV EAX,1
00401174 |.    5D             POP EBP
00401175 \.    C3             RETN
```

La sintaxi Intel és la més utilitzada en sistemes basats en Windows, de manera que la majoria d'utilitats que treballen amb els sistemes de Microsoft la sol usar.

En la llista resultant hi ha poques coses identificables a primera vista. Les diferents columnes indiquen el següent:

- Primera columna. Adreça de memòria on es troba la instrucció. Aquesta és l'adreça que es carregarà en el registre EIP per a seguir l'execució, instrucció per instrucció, del programa.

- Segona columna. Es mostren els codis en llenguatge màquina *(opcodes)* de cada instrucció, resultat de la compilació del programa.

- Tercera columna. Instruccions en llenguatge assemblador resultat de desassemblar els codis en llenguatge màquina de la segona columna.

Així, doncs, en cada línia s'observa l'adreça de memòria de la instrucció, els bytes corresponents al codi màquina i la instrucció corresponent al codi màquina en llenguatge assemblador.

S'observen, per tant, les mateixes dades que en el cas de Linux + GCC. Per a continuar l'analogia amb el cas anterior, s'observa la línia que executa la instrucció `call`:

```
00401167 |. E8 10270000  CALL EXEMPLE1.0040387C
```

Es pot veure com el sistema fa una crida a una funció, i en aquest cas no sabem si es tracta de la funció `printf`, tret que tinguem els codis de les funcions de Windows. Es produeix en aquest punt un salt en l'execució lineal del programa a l'adreça `0040387C`, que és on s'ha de trobar l'entrada a la crida a la funció `printf`.

El codi màquina indica la funció `call` mitjançant l'*opcode* `E8`, igual que en el cas anterior. Els 4 bytes restants indiquen el desplaçament. Si es té en compte que les dades s'emmagatzemen seguint la norma *little endian* es veu que el desplaçament indicat és `00002710`, que passat a decimal es correspon amb el nombre `10000`. Si es fa la resta entre l'adreça destinació i la direcció a la instrucció següent a l'execució del `call`, s'obté que:

```
Hexadecimal: 0040387C - 0040116C = 2710
Decimal: 4208764 - 4198764 = 10000
```

Obtenim el desplaçament indicat en el codi màquina. Això vol dir que la crida serà un `call near` a una adreça dins del mateix segment de codi.

A continuació s'intercala el codi font escrit en C amb el codi en assemblador:

```
Adreça            Opcodes          Assembler
int main()
{
   int x, y, result;
   00401150 >/.      55               PUSH EBP
   00401151 |.       8BEC             MOV EBP,ESP

   x = 3;
   00401153 |.       B8 03000000      MOV EAX,3

   y = 4;
   00401158 |.       BA 04000000      MOV EDX,4

   result = x + y;
   0040115D |.       03D0             ADD EDX,EAX

   printf("Resultat de x + y = %d\n", result);
   0040115F |.       8BC2             MOV EAX,EDX
   00401161 |.       50               PUSH EAX
   00401162 |.       68 28A14000      PUSH EXEMPLE1.0040A128
   00401167 |.       E8 10270000      CALL EXEMPLE1.0040387C
   0040116C |.       83C4 08          ADD ESP,8

   return(1);
   0040116F |.       B8 01000000      MOV EAX,1
   00401174 |.       5D               POP EBP
   00401175 \.       C3               RETN
}
```

S'observa que les variables locals definides no reserven espai en la pila, sinó que s'utilitzen directament registres del processador per a fer la tasca de les variables locals.

3.4.3. Comparativa dels entorns

A part del fet que els dos entorns solen utilitzar dues sintaxis diferents a l'hora de desassemblar els programes, hi ha altres diferències. La primera, i més important, és que amb compiladors diferents tenim resultats diferents en codi màquina. Així, doncs, s'ha vist que en l'exemple de Linux + GCC es reservava espai en la pila per a les variables locals, i en l'exemple de Windows + BCC aquesta reserva no es feia i s'utilitzaven els registres a manera de variables locals.

És tasca del compilador crear el codi màquina de tal manera que els resultats esperats detallats pel programa d'alt nivell s'aconsegueixin. El programador no té control sobre la manera com es crea el codi màquina final, tan sols pot confiar en la garantia que li ofereix el compilador en si. Així, doncs, si es disposa de compiladors diferents, el codi màquina resultant serà, segurament, diferent per a cadascun dels casos. Es coincideix llavors amb una norma de la programació que indica que un mateix programa (mateixes entrades i mateixes sortides) pot ser escrit d'infinites maneres diferents.

Es comparen tots dos programes:

Windows + BCC	**Linux + GCC**

Opcodes Assembler	**Opcodes Assembler**
int main()	**int main()**
{	**{**
	8d 4c 24 04 lea 0x4(%esp),%ecx
	83 e4 f0 and $0xfffffff0,%esp
	ff 71 fc pushl -0x4(%ecx)
55 PUSH EBP	55 push %ebp
8BEC MOV EBP,ESP	89 e5 mov %esp,%ebp
	51 push %ecx
	83 ec 24 sub $0x24,%esp
int x, y, result;	**int x, y, result;**
x = 3;	**x = 3;**
B8 03000000 MOV EAX,3	c7 45 f0 03000000 movl $0x3,-x10(%ebp)
y = 4;	**y = 4;**
BA 04000000 MOV EDX,4	c7 45 f4 04000000 movl $0x4,-xc(%ebp)
result = x + y;	**result = x + y;**
03D0 ADD EDX,EAX	8b 55 f4 mov -0xc(%ebp),%edx
	8b 45 f0 mov -0x10(%ebp),%eax
	01 d0 add %edx,%eax
	89 45 f8 mov %eax,-0x8(%ebp)
printf("Resultat de	**printf("Resultat de**
x + y = %d\n", result);	**x + y = %d\n", result);**
8BC2 MOV EAX,EDX	8b 45 f8 mov -0x8(%ebp),%eax
50 PUSH EAX	89 44 24 04 mov %eax,0x4(%esp)
68 28A14000 PUSH EXEMPLE1.0040ª128	c7 04 24 b0840408 movl $0x80484b0,(%esp)
E8 10270000 CALL EXEMPLE1.0040387C	e8 f7feffff call 80482d8 <printf@plt>
83C4 08 ADD ESP,8	
return(1);	**return(1);**
B8 01000000 MOV EAX,1	b8 01000000 mov $0x1,%eax

	`83 c4 24 add $0x24,%esp`
	`59 pop %ecx`
`5D POP EBP`	`5d pop %ebp`
	`8d 61 fc lea -0x4(%ecx),%esp`
`C3 RETN`	`c3 ret`
}	**}**

- Entrada a la funció principal. En Windows + BCC l'entrada a la funció principal és clarament més simple. No hi ha reserva d'espai en la pila per a les variables locals i no es fan una sèrie d'operacions, com ocorre en l'altre cas (operacions internes de control en el cas Linux + GCC). Això no té cap implicació quant a l'execució, tan sols es remarca el fet que el resultat de compilar en els dos entorns és diferent.

- Assignacions a les variables locals (`x=3`, `y=4`). En el cas Windows + BCC s'assignen els diferents valors directament a registres; en concret, els registres EAX i EDX. En el cas Linux + GCC els valors s'assignen a l'espai que tenen les variables locals en la pila. Això provoca que el codi màquina resultant en el cas Windows + BCC sigui més reduït.

- Càlcul del resultat final i assignació d'aquest resultat a una variable local (`result = x + y`). En el cas Windows + BCC es fa amb la instrucció `add` aplicada directament contra els dos registres implicats i desa el resultat en un d'aquests (*eax*). En l'altre cas, primer es carreguen els valors de les variables locals a registres, després es fa la suma i finalment es desa el valor final en l'espai assignat a les variables locals en la pila.

- Crida a la funció `printf`. En tots dos casos s'introdueixen els paràmetres necessaris en la pila per a poder fer la crida amb `call` de manera adequada. En el cas de Windows + BCC s'apilen els elements mitjançant la instrucció `push`, i en el cas Linux + GCC s'apilen els elements assignant els valors adequats dins d'un espai de la pila reservat prèviament per a fer crides a funcions. En finalitzar la crida (`call`) no es treuen els paràmetres introduïts en la pila, ja que aquest espai està reservat per a aquest propòsit. En Windows + BCC es fa necessari treure els elements prèviament introduïts, i incrementar el valor de *esp*.
  Com a detall per comentar, cal destacar que la crida en Windows + BCC es fa envers algun punt situat més cap endavant del punt actual i en el cas Linux + GCC la crida és envers algun punt anterior.

- Sortida de la funció principal del programa (`return`). En el cas Linux + GCC es fan algunes operacions internes més que en l'altre cas. Així i tot, tots dos sistemes coincideixen a assignar el valor 1 al registre *eax*, recuperar el valor anterior d'*ebp* de la pila, i executar la instrucció *ret*.

En Windows + BCC l'última instrucció és `retn`, i en Linux + GCC l'última instrucció és *ret*. Sembla que es tracti de dues instruccions diferents, encara que si s'observa l'*opcode* corresponent a la instrucció en tots dos casos, es veu que tots dos tenen assignat l'*opcode c3*. S'executa, per tant, la mateixa instrucció en tots dos casos.