Curs 07: Securitatea memoriei

Sumar curs anterior

spaţiul virtual de adrese al procesului e compus din zone statice şi dinamice, zone read-only şi read-write
în general, paginile fizice nu se alocă la comanda de alocare a utilizatorului ci la primul access: demand paging
tabela de pagini are marcată intrarea nevalidă, urmând ca informaţiile sistemului de operare să ştie că e vorba de acces nevalid sau demand paging
se fac în ordine: rezervarea unei pagini virtuale, alocarea unei pagini fizice, maparea paginii fizice la pagina virtuală
un acces la o pagină marcată nevalidă face ca MMU să transmită o excepţie de acces la pagină numită page fault, care apelează un page fault handler înregistrat de sistemul de operare
zonele read-only sunt partajate între procese
iniţial la fork() procesul părinte şi procesul copil partajează toate paginile fizice
imediat după fork(), paginile fizice sunt marcate read-only; la primul acces are loc copy-on-write: duplicarea paginii şi marcarea acesteia read-write în procesul ce a generat acţiunea de scriere
pentru a preveni epuizarea spaţiului fizic de memorie, se extinde prin folosire spaţiului de swap
swap out: evacuarea unei pagini fizice pe swap
swap in: readucerea unei pagini din swap în memoria fizică
dacă au loc operaţii de swap in şi swap out pe aceeaşi pagină, sistemul devine ineficient, apare fenomenul de thrashing
fişierele pot fi mapate în spaţiul virtual de adrese al procesului pentru eficienţă temporală şi spaţială: este cazul fişierelor executabile şi al bibliotecilor partajate

Securitatea sistemului. Securitatea memoriei

un sistem sigur funcţionează conform specificaţiilor
un defect duce la o funcţionare necorespunzătoare: abuz/atac (răuvoitor) sau eroare (greşeală)
atacatorul urmăreşte: furt de informaţie (leak, steal, disclose), îngreunare (cripple, denial-of-service) sau obţinerea controlului; obţinerea controlului pentru a fura informaţii sau îngreunare sau pentru a genera alt atac sau pentru a folosi resurse
un defect/bug este o funcţionare necorespunzătoare; dacă acest defect poate fi folosit în beneficiul atacatorului (steal, cripple, control) atunci este o vulnerabilitate; spunem că o vulnerabilitate este un defect exploatabil
un exploit este o metodă de atac a unei vulnerabilităţi
un vector de atac este o secvenţă de paşi, uzual compusă din mai multe exploit-uri care duce la obţinerea de beneficii concrete atacatorului
un subset al securităţii sistemului este securitatea aplicaţiilor

securitatea aplicaţiilor înseamnă securitatea acestora înainte de rulare (verificare, analiză statică, dinamică) şi în timpul rulării (runtime application security)
securitatea la rulare presupune atac/apărare în folosirea resurselor
un atac la rulare presupune, în general, denaturarea fluxului normal de execuţie al aplicaţie (altering the control flow graph: CGF)
securitatea memoriei aplicaţiei se referă la atacuri şi mijloace preventive legate de scrierea şi citirea datelor şi citirea şi executarea codului

Fluxul de execuţie

paşii pe care îi urmează aplicaţia în momentul rulării
fluxul de execuţie (execution flow) este definit în momentul implementării într-un limbaj de programare uzual compilat în cod maşină într-un fişier executabil care apoi este rulat din memorie într-un proces
fluxul de execuţie este descris prin graful fluxului de control al aplicaţiei (control flow graph, CFG)
un nod în graf este o secvenţă liniară de instrucţiuni (basic block)
un arc în graf este un salt (jump, branch)
în mod obişnuit, o aplicaţie este sigură dacă CFG-ul este urmat corect la rulare
este, desigur, posibil, să existe un cod de tipul backdoor, plasat de dezvoltator, parte validă din CFG care să fie declanşat într-o anumită situaţie
mai posibil, însă, un atac presupune adăugarea unor noi arce sau noduri în CFG, adică modificarea fluxul de execuţie în beneficiul atacatorului (steal, cripple, control): control flow hijack
această modificare a fluxului de execuţie se realizează prin atacuri la adresa memoriei aplicaţiei când aceasta rulează
de obicei atacurile la adresa memoriei pornesc de la o vulnerabilitate de tipul buffer overflow/index-out-of-bounds

Tipuri de zone de memorie după permisiuni

zonele de memorie pot fi: read-write, read-only, read-executable
read-executable: zone de cod/text
read-only: zonele .rodata
read-write: .data (date iniţializate), .bss (date neiniţializate), heap, stivă
un executabil (mapat în memoria procesului la load-time) conţine .text, .rodata, .data, .bss
o bibliotecă partajată (mapată la load-time sau la run-time) conţine aceleaşi zone .text, .rodata, .data, .bss
heap şi stiva (stack) sunt create la pornirea procesului
un atacator este interesat în special de zonele read-write: ce poate să suprascrie pentru a afecta fluxul de execuţie şi a obţine beneficii (steal, cripple, control)
un atacator se poate folosi de codul existent în zonele read-executable: code reuse

Reminder de la IOCLA: Buffer overflow. Index out of bounds

bufferele sunt zone continue de memorie care pot fi scrise şi citite: sunt plasate în zone read-write
un buffer: adresă de start, dimensiune
un buffer este definit în C ca un array: un şir de caractere e un caz particular de array
în special în lucrul cu şiruri, folosirea necorespunzătoare a bufferelor poate duce la suprascrierea de date
buffer overflow: parcurgerea buffer-ului element cu element şi trecerea de limita superioră (apeluri de tipul memcpy, strcpy, fgets pot face asta)
index out of bounds: accesarea unui index din afara spaţiului buffer-ului (index negativ sau dincolo de dimensiunea buffer-ului)
putem spune că buffer overflow e un subcaz de index out of bounds
cauzate de erori de programare: char v[32]; fgets(v, 64, stdin);
se pot suprascrie date care afectează fluxul de execuţie şi oferi beneficii atacatorului
în general nu se primeşte segmentation fault sau alt tip de excepţii pentru că paginile accesate prin overflow/index out of bounds sunt valide în spaţiul virtual de adrese al procesului
+ demo cu buffer overflow care nu generează crash
în Java sau alte limbaje se primeşte excepţie pentru că maşina virtuală face verificările, cu dezavantajul de overhead temporal

Reminder de la IOCLA: Funcţionarea stivei

bufferele din funcţii sunt alocate pe stivă (mai puţin cazul când sunt declarate statice); sunt dese atacuri ce folosesc vulnerabilităţi de tipul buffer overflow şi index out of bounds pe stivă
stiva conţine cadre de stivă (stack frames) pentru fiecare funcţie apelată
structura unui cadru de stivă depinde de convenţia de apel (calling convention)
pe x86 un cadru de stivă conţine:
* parametrii funcţiei
* adresa de retur
* frame pointer (ebp)
* variabile locale
+ diagramă cu set de cadre de stivă
este interesant să realizăm atacuri pe stivă pentru că sunt informaţii critice ce pot fi suprascrise, în special adresa de retur
dacă atacatorul exploatează o vulnerabilitate de tipul buffer overflow sau index out of bounds, atunci poate suprascrie adresa de retur şi poate redirecta/deraia (hijack) fluxul de control al programului în beneficiul său
buffer overflow-urile pe stiva se numesc stack buffer overflow; există şi heap buffer overflow care suprascriu informaţii din heap

Atacuri simple de memorie. Code pointeri

pentru a deraia fluxul de execuţie al unei aplicaţii, un atacator urmăreşte suprascrierea unor pointeri la zone de cod, precum adresa de retur
suprascrierea adresei de retur cu o valoare convenabilă va redirecta instruction pointer-ul la acea adresă de unde atacatorul va folosi codul dorit
code pointers sunt zone de memorie ce conţin adrese de zone de cod: adresa de retur, pointeri de funcţie
din perspectiva apărătorului, aceşti pointeri trebuie protejaţi să nu fie suprascrişi; atacatorul doreşte să îi suprascrie
din suprascrierea unui code pointer rezultă două tipuri de atacuri:
* code reuse: suprascrierea cu o adresă deja existentă de cod (.text sau .text dintr-o bibliotecă)
* code injection: suprascrierea cu un cod scris într-o zonă read-write şi executarea codului scris în acea zonă
avantaj code reuse: mai simplu, foloseşte resurse existente
avantaj code injection: flexibil, se poate injecta ce cod se doreşte
+ demo cu suprascriere pointer de funcţie
+ demo cu Stack buffer overflow pentru suprascriere pointer de funcţie
+ demo cu Stack buffer overflow pentru suprascriere adresă de retur

Shellcode

o secvenţă de cod maşină injectată pentru a fi executată: code injection
uzual este combinată cu exploatarea unui buffer overflow şi suprascrierea unui code pointer pentru a ajunge la acea zonă
pentru a fi executată zona trebuie să fie read-write (să poată fi scris shellcode-ul) şi executabilă (să poată fi executat)
convenţional un shellcode deschide un shell (exec("/bin/sh")), dar poate fi folosit la orice: deschis un socket, schimbat permisiuni, citit un fişier
un shellcode conţine cod maşină şi foloseşte apeluri de sistem, nu apeluri de funcţii de bibliotecă; nu ar şti unde este plasat în memorie şi unde se găsesc adresele funcţiilor
+ exemplu de shellcode (în limbaj de asamblare)
+ demo cu Stack buffer overflow cu shellcode
+ demo cu Stack buffer overflow cu shellcode pe stivă

Mecanisme defensive

pentru ca un shellcode să ruleze este nevoie de:
* o vulnerabilitate (precum buffer overflow) care să permită suprascrierea unui code pointer
* o intrare în program care să ducă la citirea shellcode-ului într-o zonă read-write
* executarea shellcode-ului din zona în care a fost suprascris
metodele dinainte de deploy pot duce la eliminarea vulnerabilităţilor

ne referim în particular la metode din momentul rulării aplicaţiei (runtime application security)
în general se urmează un pattern: atac, soluţie defensivă, bypass
input validation: valiarea intrării pentru a nu permite date binare acolo unde ar fi nevoie de text; bypass: shellcode alfanumeric
stack guard, stack canary, stack smashing protection: plasarea unei valori între buffer şi adresa de retur
safe stack: plasarea code pointerilor şi a datelor critice pe o stivă dedicată
DEP (data execution prevention): o zonă read-write nu poate fi şi executabilă
ASLR (address space layout randomization): zonele de memorie sunt plasate la adrese aleatoare şi nu se poate uşor găsi adresa cu care să se fie suprascrie un code pointer
CFI (control flow integrity): asigurat că este respectat fluxul de execuţie al programului şi nu se adaugă noi noduri sau arce în CFG

Stack Guard

Stack Smashing Protection (SSP) sau stack canary
se plasează o valoare între buffer şi adresa de retur
suprascrierea adresei de retur prin buffer overflow va însemna suprascrierea valorii canar, lucru ce va fi detectat la părăsirea funcţiei
canarul este plasat într-o zonă dedicată
se pot suprascrie în continuare variabile locale
+ demo cu program compilat cu SSP şi fără SSP, văzut codul în limbaj de asamblare
mic cost de performanţă, se poate aplica SSP selectiv pe funcţiile ce conţin pointeri
bypass: se suprascrie canarul cu el însuşi; se plasează uzual 0x00 şi 0x0a în canar pentru a "opri" funcţii de lucru cu şiruri din suprascriere
bypass: se suprascrie handle-ul de tratare a suprascrierii canarului
+ demo cu stack canary bypass

AddressSanitizer

https://github.com/google/sanitizers/wiki/AddressSanitizer
soluţie integrată de securizare a memoriei
overhead semnificativ, bun în faza de dezvoltare
mai multe tipuri de "sanitizers"
integrat în compilator

Safe Stack

code pointerii sunt plasaţi într-o zonă dedicată: safe stack
buffer overflow-ul nu suprascrie code pointeri
se modifică stack frame-ul unei funcţii
e nevoie de modificarea modului în care compilatorul generează codul de tip prologue şi epilogue al funcţiilor

Data Execution Prevention

atacurile cu shellcodes (de code injection) sunt posibile dacă zonele read-write pot fi şi executate
există suport în hardware pentru marcarea paginilor ca neexecutabile
stiva, heap-ul, data, bss sunt marcate ca neexecutabile: se poate injecta cod, nu se poate executa
bypass: se foloseşte alt tip de atac (code reuse) care să remapeze o zonă read-write ca executabilă (folosind un apel de tipul mprotect()/VirtualProtect())

Code Reuse. Return-to-libc

refolosirea codului existent în memorie în zonele read-execute: .text sau .text din biblioteci
se pot folosi funcţii întregi sau părţi din funcţii
return-to-libc însemnă apelul unei funcţii din biblioteca standard C; un apel uzual este system("/bin/sh"); pentru deschiderea unui shell
variaţii de code reuse sunt return-oriented programming (înlănţuirea de secvenţe mici care se încheie în ret) sau jump-oriented programming (înlănţuirea de secvenţe mici care se încheie în instrucţiuni precum jmp *eax)
trebuie ştiute adresele de cod unde se face saltul

ASLR

măsură defensivă care face dificilă descoperirea de adrese: adrese de cod, adrese în stivă
se plasează aleator zone din spaţiul de adrese al procesului: heap, stivă, biblioteci
+ demo cu ASLR activat şi dezactivat
dacă un executabil este compilat cu suport de PIE (Position Independent Executable) se plasează aleator şi zonele executabilului (.text, .rodata, .data, .bss)
+ demo cu executabil cu PIE şi fără PIE
bypass: pe sistemele pe 32 de biţi se poate face brute force până când se "nimereşte" adresa
bypass: memory disclosure: se obţine (leak) o adresă care ajută la calculul altor adrese

CFI

validarea fluxului de execuţie şi întreruperea programului dacă apar arce sau noduri noi în CFG
overhead semnificativ, util în anumite situaţii sau în etapa de testare
bypass: data-oriented attacks: atacuri care folosesc abuziv fluxuri existente în CFG dar care nu ar trebui permise
ideal este ca în faza de dezvoltare/testare să se acopere cât mai mult din CFG-ul programului (CFG coverage)

Mai multe detalii

Security Summer School: http://security.cs.pub.ro/summer-school/wiki
Compilatoare (C3, anul 4 semestrul 1): https://ocw.cs.pub.ro/courses/cpl
Computer and Network Security: http://ocw.cs.pub.ro/cns
Wargame/CTF sites: http://captf.com/practice-ctf/

Sumar

un sistem este sigur dacă funcţionează conform specificaţiilor
o componentă este securitatea aplicaţiilor la rulare (runtime application security)
o parte importantă este securitatea memoriei
spaţiul virtual de adrese al procesului este compus din zone de memorie cu permisiuni diferite: read-write, read-only, read-executable
o aplicaţie are un flux de execuţie descris de control flow graph (CFG)
există două tipuri de atacuri: atacuri ce adaugă noi noduri în graf (code injection attacks) sau care adaugă arce şi refolosesc graful în moduri benefice atacatorului (code reuse attacks)
vulnerabilităţile “standard” sunt buffer overflow şi out of bounds errors
un atacator urmăreşte suprascrierea de informaţii pentru a modifica CFG-ul programului
interesant este de realizat stack buffer overflow attacks, pentru că se suprascrie adresa de retur
adresa de retur este un code pointer, un pointer la o zonă de cod; suprascrierea unui code pointer oferă atacatorului posibilitatea controlului fluxului de execuţie (control flow hijack)
partea de code injection presupune injectarea unei secvenţe de cod în memoria procesului: shellcode
mecanisme de protecţie sunt: stack guard, data execution prevention, address space layout randomization, control flow integrity
în general se urmăreşte schema: vulnerabilitate/problemă, atac, metodă preventivă, bypass la metodă preventivă