Securitatea memoriei

SO: Curs 07

Cuprins

- Procese și executabile
- De la proces la executabil
- Alterarea spațiului de adresă
- Vulnerabilități și atacuri
- Exploatarea memoriei
- Metode ofensive și mecanisme defensive

Suport de curs

- Jon Erickson - Hacking: The Art of Exploitation, 2nd Edition
  - Section 0x270. Memory Segmentation
  - Chapter 0x300. Exploitation
- https://io.netgarage.org/
- Aleph One – Smashing the Stack for Fun and Profit
- Understanding the Stack (CMSC 311, Computer Organization)

PROCESE ȘI EXECUTABILE

Procese

- Program în execuție
- Entitate planificabilă
- CPU, memorie, I/O
- Descriptori de fișier, semnale
- Cuantă de timp, prioritate, stare
- Spațiu virtual de adrese, zone de memorie
  - date (variabile), zone read+write
  - cod (instrucțiuni), zone read+executable

Spațiul virtual de adrese

- Spațiu de adresare unic procesului
- Toate informațiile de memorie (cod, date)
- Adrese virtuale, pagini virtuale
  - mapare peste pagini fizice
- Programatorul lucrează doar cu adrese virtuale

Spațiul virtual de adrese (2)

```
$ pmap -p $(pidof exec-addr)
6293:   ./exec-addr
0000000000400000      4K r-x-- /home/exec-addr
0000000000600000      4K rw--- /home/exec-addr
00007fa6d730f000   1664K r-x-- /lib/x86_64-linux-gnu/libc-2.18.so
00007fa6d74af000   2044K ----- /lib/x86_64-linux-gnu/libc-2.18.so
00007fa6d76ae000     16K r---- /lib/x86_64-linux-gnu/libc-2.18.so
00007fa6d76b2000      8K rw--- /lib/x86_64-linux-gnu/libc-2.18.so
00007fa6d76b4000     16K rw---   [ anon ]
00007fa6d76b8000    128K r-x-- /lib/x86_64-linux-gnu/ld-2.18.so
00007fa6d789c000     12K rw---   [ anon ]
00007fa6d78d3000     16K rw---   [ anon ]
00007fa6d78d7000      4K r---- /lib/x86_64-linux-gnu/ld-2.18.so
00007fa6d78d8000      4K rw--- /lib/x86_64-linux-gnu/ld-2.18.so
00007fa6d78d9000      4K rw---   [ anon ]
00007fffd9b4d000    132K rw---   [ stack ]
00007fffd9bfe000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total             4068K
```

Crearea unui proces

- Dintr-un proces existent
- CreateProcess() pe Windows
- fork() + exec() pe Linux
- La exec() se înlocuiește imaginea de executabil
  - Spațiul de memorie (date, cod) este înlocuit
- La exec() informațiile sunt preluate dintr-un fișier în format de executabil

Fișiere în format executabil

- Obținute din compilarea surselor și link-editarea modulelor obiect
- Descrierea zonelor de memorie: date și cod
- Simbolurile definite în program
  - variabile
  - funcții
  - un simbol cuprinde: nume, tip, adresă, spațiu ocupat

Formate de executabile

- ELF: Formatul standard pe Unix
- PE: Windows
- Mach-O: Mac OS X
- Listare secțiuni

  objdump --headers <executable-file>
- Listare simboluri

  objdump --syms <executable-file>
- Dezasamblare (zone cu instrucțiuni)

  objdump --disassemble <executable-file>

DE LA EXECUTABIL LA PROCES

Fazele prin care trece un program

- Compile-time
  - se creează modul obiect dintr-un fișier cod sursă
- Link-time
  - se creează fișier executabil din module obiect și biblioteci
- Load-time
  - se creează un proces dintr-un executabil
- Run-time
  - procesul execută acțiuni definite în codul său

Maparea zonelor de executabil

- .text, .data, .rodata, .bss
- Se întâmplă la load-time
- Se folosește un apel de forma mmap pentru a mapa zonele de cod și date în memoria procesului
- În Linux se pot observa folosind utilitarul pmap

Maparea bibliotecilor dinamice

- .so pe Linux, .dll pe Windows, .dylib pe Mac OS X
- Au tot format de executabil
- Au date și cod
- Maparea este similară dar ...
  - pot fi mapate la adrese diferite depinzând de executabil
  - sau chiar la adrese diferite la diferite rulări ale procesului
- PIC: Position Independent Code

ASLR

- Codul și datele bibliotecilor sunt mapate la adrese diferite la fiecare rulare
- Rațiuni de securitate
- Address Space Layout Randomization
- Poate fi observat folosind utilitarul pmap
- ... sau folosim ldd cu argument un executabil

ALTERAREA SPAȚIULUI DE ADRESĂ LA RUN-TIME

Stiva

- Este folosită pentru a reține informații legate de apelurile de funcții
- “Stack frame” creat pentru fiecare funcție
- Un stack frame conține informații despre apelant (caller) și apelat (callee)
  - parametrii funcției
  - adresa de retur
  - fostul frame pointer
  - variabile locale

Alocarea pe stivă

- Pentru variabile locale unei funcții
- Dinamică, la run-time
- Automată (alocarea și dezalocarea)
  - alocare la intrarea în funcție/bloc
  - dezalocarea la ieșirea din funcție/bloc
- Se mai alocă, implicit, valoarea de retur, fostul frame pointer și parametrii funcției

Heap-ul

- Alocare dinamică, la run-time
- Apelurile malloc/free
- Atenție la
  - memory leak-uri
  - omiterea apelării free
  - dangling pointers
  - double free

Maparea memoriei

- Alocare dinamică, la run-time
- mmap, VirtualAlloc (Windows)
- Permite mapare de fișiere, partajare de memorie
- Control al alocării/dezalocării
- Permisiuni de acces
- Granularitate la nivel de pagină

VULNERABILITĂȚI ȘI ATACURI LA MEMORIA UNUI PROCES

Execuția codului într-un proces

- Zone de cod
- Instruction pointer
  - poziția curentă
  - incrementat cu dimensiunea unei instrucțiuni
- Fluxul se schimbă la branch-uri sau apeluri de funcții
- Pointeri de funcții

Bug-uri și vulnerabilități

- Ce este un bug?
- Când este un bug o vulnerabilitate?
- Ce obiective sunt în exploatarea unei vulnerabilități?

Tipuri de vulnerabilități la nivelul memoriei

- Suprascrierea datelor cu date de atac
  - suprascriere variabile (verificate în if)
  - suprascriere pointeri de funcții
- Alterarea fluxului de execuție
- Executarea de apeluri arbitrare
- Executarea de cod arbitrar (code injection)

exec(“/bin/bash”)

- Sau echivalentul system(“/bin/bash”)
- Obținerea unui shell într-o aplicație existentă
- Obiectivul inițial al unui atac
- Atacatorul se va folosi de vulnerabilități ale memoriei pentru a porni un shell
- Ulterior:
  - obținere informații confidențiale
  - denial of service
  - privilege escalation

STACK BUFFER OVERFLOW

Injectare de cod

- Punere de cod într-o zonă writable și executarea sa
- Să putem scrie cod
  - Folosim funcții de citire input (fgets, scanf)
- Să putem executa cod de acolo
  - zonă executabilă
  - jump la acea adresă
- E nevoie de o zonă simultan writable + executable

Shellcode

- Instrucțiuni în cod binar
- Se scriu într-o zonă / buffer
- Se “sare” aici pentru execuție
- De obicei realizează exec(“/bin/bash”)
- Se încearcă injectarea acestuia într-o zonă accesibilă

Structura stivei (reminder)

- Adresă de retur
- Fostul frame pointer
- Variabile locale (buffere incluse)

- Dacă un buffer este neîncăpător (overflow) putem suprascrie date, eventual chiar adresa de retur

Stack buffer overflow

- Scrierea în buffer peste dimensiunea alocată
- De ce merge?
  - avem spațiu alocat pe stivă
- Ce suprascriem?
  - alte variabile
  - adresa de retur
- Ce obținem
  - alterarea fluxului de execuție
  - rulare de cod arbitrar

Suprascrierea adresei de retur

- Adresa pe care o va folosi apelatul la încheierea funcției
  - este adresa instrucțiunii următoare față de cea folosită de apelant în momentul apelului
- Se găsește pe stivă
- Stack buffer overflow poate conduce la suprascrierea adresei de retur
- Se pune adresa shellcode-ului
- Shellcode-ul poate sta pe stivă

Funcții de lucru cu șiruri

- str*
- Împreună cu funcțiile de citire de intrare (fgets) sunt principalele funcții exploatabile
- NBTS (NUL-terminated byte strings)
  - dacă un șir nu e NUL-terminat, putem suprascrie dincolo de dimensiunea sa
- Lungimea unui șir
  - trebuie știută tot timpul
  - presupunei legate de lungimea șirului pot conduce la vulnerabilități de securitate

METODE OFENSIVE ȘI MECANISME DEFENSIVE

Injectare de cod

- Zone writable + executable
- Se poate plasa cod acolo și executa
- Acest cod se numește “shellcode”
- Clasic: pe stivă + stack buffer overflow
  - citire buffer de la standard input
  - variabilă de mediu

DEP

- Data execution prevention
- O zonă writable nu este executable
- W^X
- NX flag pe arhitecturile moderne
- Bypass: apel mprotect, VirtualProtect

return-to-libc

- Executarea unei funcții existente
- Clasic, system(“/bin/bash”)
- Nu este nevoie de injectare de cod
- Trebuie știută adresa funcției

Canary value

- Plasată pe stivă între variabile locale și adresa de retur
- Se verifică coerența informației la părăsirea funcției
- Previne efectul stack buffer overflows pentru a suprascrie adresa de retur

Suprascriere de alte date

- Nu suprascriem adresa de retur
- Variabile locale alterează fluxul de lucru
- Pointeri de funcții
- Suprascriere pointeri de gestiune a heap-ului

ASLR

- Randomizează plasarea bibliotecilor
  - dificil de găsit adresa funcțiilor de bibliotecă
  - dificil de realizat return-to-libc
- Randomizează plasarea stivei
  - adresa de retur se suprascrie cu o adresă efectivă, nu cu o funcție cu jump relativ
  - adresa buffer-ului de pe stivă variază la fiecare rulare
  - dificil de știut unde să sari pe stivă

CONCLUZII

Safe coding

- Atenție la funcții de lucru pe șiruri
- Folosiți suport DEP, ASLR, canary value
- Analiză statică pentru verificarea de bug-uri sau vulnerabilități

Cuvinte cheie

- proces
- spațiu virtual
- executabil
- biblioteci
- pmap
- zone de memorie
- ASLR
- load-time
- aun-time
- alocare la run-time
- stivă
- stack frame
- heap
- bug
- vulnerabilitate
- exec(“/bin/bash”)
- injectare de cod
- stack buffer overflow
- alterarea fluxului
- shellcode
- adresa de retur
- funcții pe șiruri
- return-to-libc
- DEP
- canary value
- safe coding