Lucrare practica / Lucrare de laborator nr. 2 (p/u Linux)

1. Introducere

Ce este analiza dinamică a software-ului: Analiza Dinamică a Software-ului (DAST – *Dynamic Application Security Testing*) reprezintă testarea securității unei aplicații **în timp ce aceasta rulează**. Altfel spus, DAST implică **executarea aplicației** și interacțiunea cu ea (trimiterea de input-uri, simularea atacurilor) pentru a observa comportamentul și a identifica eventuale vulnerabilități. Acest tip de analiză se desfășoară din perspectiva unui utilizator extern (sau a unui atacator), fără a avea acces la codul sursă – se testează "cutia neagră". Rezultatele DAST evidențiază probleme de securitate care apar **doar în execuție**, cum ar fi erori de memorie, injecții de cod, comportamente neașteptate la anumite input-uri etc.

Analiza statică vs. analiza dinamică: Analiza statică (cunoscuta și ca SAST – *Static Application Security Testing*) se realizează *fără a rula codul*, examinând codul sursă sau binarul aplicației "la rece" pentru a detecta vulnerabilități potențiale. De exemplu, un analizor static poate parcurge codul C/C++ pentru a detecta posibilități de *buffer overflow* sau dereferințe ilegale de pointer, **înainte** de a compila și executa programul. Prin contrast, analiza dinamică (DAST) presupune rularea efectivă a aplicației și simularea de atacuri asupra acesteia, identificând vulnerabilități manifestate doar la runtime. Cu alte cuvinte, SAST "citește" codul, pe când DAST "atacă" aplicația rulând. Ambele abordări sunt complementare: analiza statică poate găsi probleme de securitate încă din faza de dezvoltare (ex. variabile neinițializate, injecții SQL în cod, erori logice), pe când analiza dinamică descoperă breșe în versiunea *deployată* a aplicației, în mediul său de execuție real.

Tipuri de vulnerabilități detectate prin DAST: Prin testare dinamică se pot identifica o gamă largă de vulnerabilități de securitate, printre care:

- **Vulnerabilități web de intrare/ieșire nevalidată:** injecții SQL, **cross-site scripting (XSS)**, includeri de fișiere, cross-site request forgery (CSRF) etc. Acestea sunt detectate prin instrumente de scanare web (ex: Burp Suite, OWASP ZAP) care trimit payload-uri malițioase aplicației web și monitorizează răspunsurile.
- **Probleme de gestiune a memoriei:** *buffer overflow*, *use-after-free*, *double free*, **scurgeri de memorie** (memory leaks) etc., întâlnite în aplicații native (C/C++). Acestea pot fi evidențiate de unelte precum Valgrind sau AddressSanitizer rulând programul și monitorizând operațiile de memorie.

- **Erori de logică sau de flux de execuție:** situații în care aplicația cade (crashează) sau intră în stări neașteptate atunci când primește input-uri neprevăzute. Tehnici de *fuzzing* (ex: AFL) pot descoperi astfel de erori generând automat sute/mii de input-uri diverse și verificând dacă aplicația se blochează sau se comportă anormal.
- **Vulnerabilități de configurare sau dependențe:** de exemplu versiuni nesigure ale unor biblioteci, configurări greșite ale serverului web etc., pot fi de asemenea detectate de scanerele dinamice (unele vor raporta software-ul serverului și versiunile pentru a semnala cunoscute probleme).
- **Probleme de concurență sau performanță la execuție:** unele instrumente dinamice specializate (ex: Valgrind – modul Helgrind, sau instrumente de profiling) pot evidenția condiții de cursă (race conditions), blocări inter-thread (deadlocks) sau consum excesiv de resurse în timpul rulării.

Rezumat: DAST testează *practic* securitatea unei aplicații rulând-o cu date de test și observând direct efectele. Această introducere acoperă conceptele cheie și pregătește terenul pentru configurarea mediului și utilizarea efectivă a instrumentelor de analiză dinamică în continuare.

2. Pregătirea mediului

Înainte de a începe testele dinamice, este necesară pregătirea mediului de lucru – atât hardware, cât și software – și instalarea instrumentelor necesare pe platforma dorită (Windows sau Linux). Această secțiune va detalia cerințele și pașii de instalare pentru uneltele folosite în laborator: **Valgrind**, **AFL (American Fuzzy Lop)**.

Cerințe hardware și software: Pentru analiza dinamică, configurația hardware obișnuită este de regulă suficientă, însă există câteva aspecte de luat în considerare:

- Un procesor multicore și memorie RAM adecvată: unele unelte de fuzzing (AFL) pot beneficia de rulare paralelă pe mai multe nuclee, iar scanerele web pot consuma memorie când analizează site-uri mari. Un sistem modern cu 4-8 GB RAM și CPU dual/quad-core este, în principiu, suficient pentru scenariile de laborator.
- Spațiu pe disc: fuzzing-ul poate genera numeroase fișiere (cazuri de test), iar log-urile Burp/ZAP pot ocupa spațiu. Asigurați-vă că aveți câteva sute de MB liberi disponibili.

Sistem de operare: **Linux** este recomandat pentru instrumente ca Valgrind și AFL (care au suport nativ excelent pe Linux). **Windows** va fi folosit mai ales pentru instrumente de testare web

(Burp, ZAP) – acestea sunt multi-platformă, deci pot rula și pe Linux, însă în practică multe testări web se fac de pe stații Windows.

Instalarea Valgrind: Valgrind este disponibil preponderent pe Linux. Pași de instalare:

- **Linux (Ubuntu/Debian):** Valgrind se găsește în repository-urile majorității distribuțiilor. Rulați comanda apt pentru instalare:

  ```
  bash
  sudo apt-get update && sudo apt-get install valgrind
  ```

  După instalare, verificați versiunea cu `valgrind --version`. Valgrind include mai multe utilitare, cel mai folosit fiind **Memcheck** (pentru detectarea erorilor de memorie).

- **Windows:** Nu există o versiune nativă Valgrind pentru Windows. Dacă aveți nevoie de funcționalități similare (detectare leak-uri, erori de memorie) pe Windows, puteți folosi alternative precum **Dr. Memory** sau rularea programelor în WSL/VirtualBox cu Valgrind. În contextul acestui laborator, recomandarea este să rulăm programele C/C++ în mediul Linux (nativ sau VM) pentru a folosi Valgrind.

Instalarea AFL (American Fuzzy Lop): AFL este un fuzzer orientat pe securitate, inițial dezvoltat pentru Linux.

- **Linux:** AFL poate fi instalat din surse sau din pachetele distribuției. Pe distribuții Debian/Ubuntu recente, se poate instala pachetul *afl*:

  ```
  bash
  sudo apt-get install afl++  # instalare AFL++ (continuarea open-source
  a AFL)
  ```

  (Notă: AFL original a evoluat în AFL++, pachetul poate fi numit afl++.)
  Alternativ, se poate construi din codul sursă (disponibil pe GitHub) prin rularea clasică

3. Utilizarea fiecărui instrument pentru analiza dinamică

În această secțiune vom explora **cum se folosește concret fiecare instrument** menționat, pentru a identifica vulnerabilități sau probleme de securitate. Vom parcurge pe rând:

- *Valgrind* – detectarea problemelor de memorie în aplicații C/C++.
- *AFL (American Fuzzy Lop)* – fuzzing orientat pe descoperirea erorilor de securitate.

Fiecare sub-secțiune va descrie succint **scopul instrumentului**, apoi va oferi **comenzi și pași de utilizare**, ilustrând cum se poate integra unealta în procesul de testare dinamică.

Valgrind

Descriere: Valgrind este o suită de instrumente de analiză a programelor, cel mai cunoscut fiind **Memcheck**, folosit pentru a detecta erori de memorie în programele C/C++. Cu Valgrind putem afla dacă un program citește sau scrie în afara limitelor unui buffer, dacă folosește variabile neinitializate, dacă uită să elibereze memorie alocată (memory leaks) etc. Practic, Valgrind rulează programul într-un mediu virtualizat/instrumentat, monitorizând toate operațiile de memorie.

Cum se utilizează: presupunând că avem un executabil compilat (cu debug symbols preferabil, flag `-g` la compilare, pentru a obține informații mai utile despre locațiile erorilor):

1. **Lansarea programului sub Valgrind:** Folosim comanda `valgrind` urmată de opțiuni și executabil. Cea mai folosită opțiune este `--leak-check=full` pentru a activa raportul complet de leak la terminare. Exemplu:

   ```
   bash
   valgrind --leak-check=full ./app_vulnerabila arg1 arg2
   ```

   unde `./app_vulnerabila` este programul nostru (putem trece și argumente dacă e nevoie). Valgrind va porni programul și va începe să monitorizeze.

2. **Interpretarea rulării sub Valgrind:** În timpul execuției, dacă apare o eroare de memorie (de ex. acces ilegal), Valgrind va întrerupe fluxul și va afișa un mesaj de eroare. La finalul execuției, indiferent dacă apar sau nu erori în timpul rulării, Valgrind va afișa un **rezumat al memoriei**. Un exemplu de ieșire Valgrind pentru un program cu probleme ar putea arăta astfel:

   ```
   plaintext
   Copy
   ==1234== Invalid read of size 4
   ==1234==    at 0x401152: main (program.c:10)
   ```

```
==1234==  Address 0x5f5f5f5f5f is not stack'd, malloc'd or (recently)
free'd
...
==1234== HEAP SUMMARY:
==1234==    in use at exit: 8 bytes in 1 blocks
==1234==  total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==1234==
==1234== LEAK SUMMARY:
==1234==    definitely lost: 8 bytes in 1 blocks
==1234==    possibly lost: 0 bytes in 0 blocks
==1234==    still reachable: 0 bytes in 0 blocks
==1234==         suppressed: 0 bytes in 0 blocks
==1234==
==1234== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

În acest exemplu (simulat), Valgrind raportează o **citire invalidă** (Invalid read) la o anumită adresă, indicând linia din cod (`program.c:10`) unde a avut loc, și un **leak de 8 bytes** neeliberați (definitely lost: 8 bytes). Secțiunea *HEAP SUMMARY* arată că programul a alocat 8 bytes și nu i-a eliberat. Astfel de informații ajută programatorul să identifice exact unde are loc eroarea și ce memorie nu a fost eliberată.

3. **Opțiuni utile:** Pe lângă `--leak-check=full`, alte opțiuni frecvent folosite:
   - `--show-leak-kinds=all` – pentru a vedea toate tipurile de leak (inclusiv "indirect" leaks, etc.).
   - `--track-origins=yes` – foarte util pentru erori de utilizare a memoriei neinitialize, arată originea variabilelor neinitialize.
   - `-v` – mod verbose, oferă informații detaliate (poate fi mult output).
   - `--log-file=valgrind_out.txt` – pentru a salva rezultatele într-un fișier în loc de consola standard.
4. **Exemplu simplu de problemă depistată:** Dacă programul are, de exemplu, o funcție care alocă memorie cu `malloc` și nu o eliberează înainte de terminare, Valgrind va raporta acea memorie la *definitely lost* în Leak Summary. Dacă programul accesează un array dincolo de limita sa, Valgrind va raporta *Invalid read/write* și va indica adresa accesată ilegal. Astfel, folosind Valgrind, un dezvoltator poate rula aplicația cu diverse scenarii de input și poate detecta imediat eventualele erori de memorie, putând apoi să le corecteze.

Concluzie Valgrind: Folosiți Valgrind pe orice program C/C++ pe care îl testați dinamic. Este foarte simplu de rulat și oferă indicii valoroase despre probleme grave de memorie care altfel ar fi greu de depistat. În contextul securității, erorile de memorie precum cele găsite de Valgrind (buffer overflow, use-after-free) pot reprezenta vulnerabilități exploatabile – deci identificarea și remedierea lor este esențială.

AFL (American Fuzzy Lop) (15 min)

Descriere: AFL este un fuzzer de tip *coverage-guided*, orientat pe descoperirea automată a bug-urilor (în special a celor de securitate) prin generarea inteligentă de input-uri de test. AFL instrumentează programul țintă și încearcă multiple variații de date de intrare, observând ce ramuri de cod sunt parcurse (de aici *coverage guided* – își adaptează input-urile ca să acopere cât mai mult cod). Când programul țintă se blochează sau are un comportament anormal (ex: un crash), AFL salvează acel caz de test pentru analiză.

Pregătirea pentru fuzzing cu AFL: AFL necesită acces la codul sursă al programului (pentru a-l compila cu un instrumentator special) sau, alternativ, poate lucra pe binare folosind QEMU mode (dar asta e avansat). Pentru laborator, vom fuzza un program cunoscut (sau scris de noi) – pașii sunt:

1. **Compilarea programului cu afl-gcc/afl-clang:** AFL vine cu un wrapper pentru compilator. Exportați variabila de mediu pentru a folosi AFL:

   ```
   bash
   export AFL_USE_ASAN=1   # opțional, pentru a combina cu
   AddressSanitizer
   afl-clang-lto -g -O0 -o program_vuln program_vuln.c
   ```

   (Notă: folosim -g pentru simboluri debug și -O0 sau -O2 moderat, evitând -O3 care complică afl).

   Dacă AFL este instalat ca afl++, comanda poate fi `afl-cc` sau `afl-clang-fast`. După compilare, vom obține un executabil `program_vuln` instrumentat de AFL (conține cod suplimentar pentru a raporta execuția).

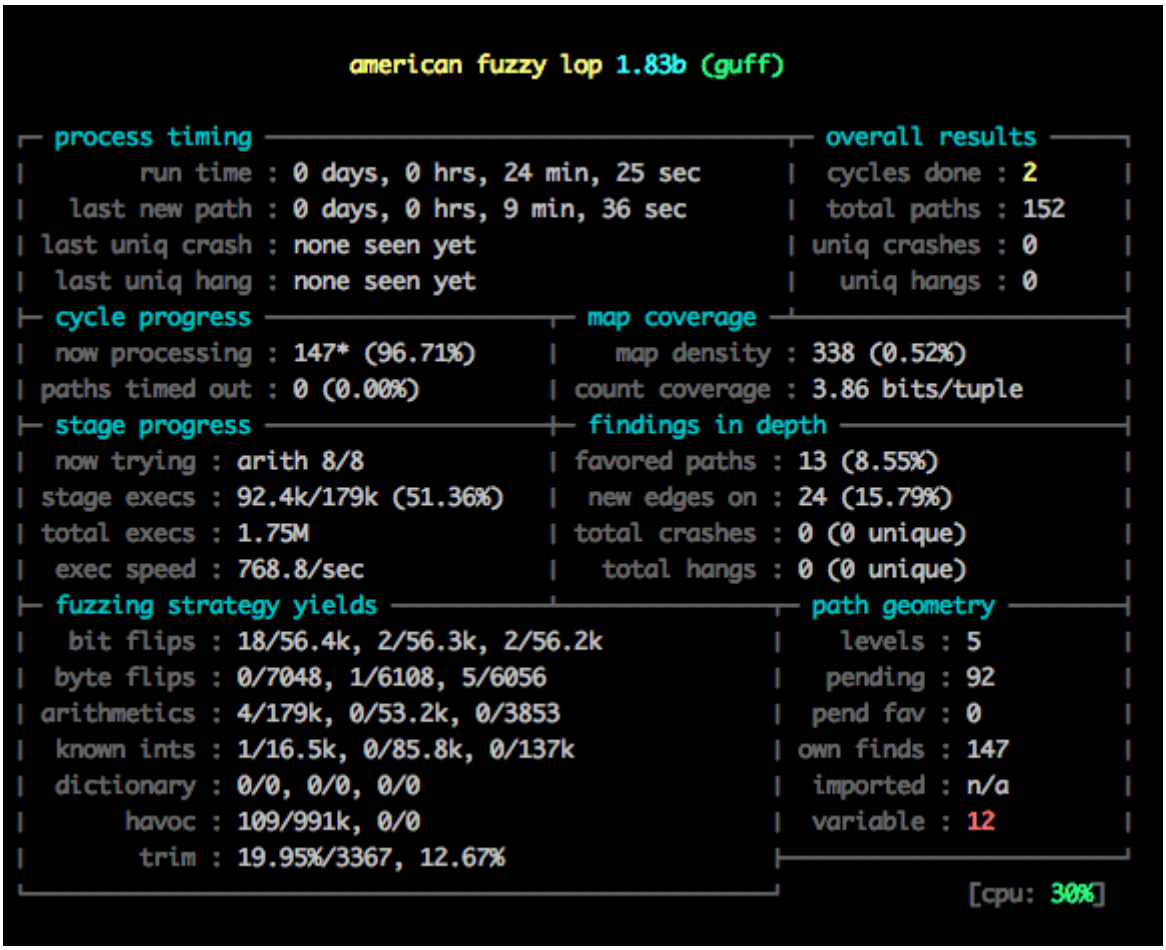
   Pregătire input-uri inițiale: AFL are nevoie de un director cu una sau câteva intrări de pornire (*seed inputs*). Creăm un director, de exemplu `inputs/`, și punem acolo fișiere de test de bază. Chiar și un fișier gol sau cu un caracter poate fi suficient ca seed minimal.

2. **Lansarea AFL pentru fuzzing:** Comanda principală:

```
bash
afl-fuzz -i inputs/ -o outputs/ -- ./program_vuln @@
```

Explicație: `-i inputs/` specifică directorul cu fișierele seed, `-o outputs/` este directorul unde AFL va salva rezultatele (fișierele generate, crash-urile, log-uri). `--` delimitează opțiunile AFL de cele ale programului: `./program_vuln @@` indică programul țintă și argumentul la linia de comandă unde AFL va introduce calea fișierului de test (AFL înlocuiește `@@` cu calea fiecărui fișier generat). Dacă programul citește input de la stdin și nu din fișier, omiterea `@@` face ca AFL să pipe-uiască datele prin stdin.

Odată pornit, AFL va afișa în terminal interfața sa text cu statusul testării.



Interfața AFL în timpul unui proces de fuzzing (text UI în consolă) – afișează statistici despre progres: număr de cazuri de test generate, căi unice acoperite, crash-uri descoperite etc.

3. **Monitorizarea și rularea fuzzing-ului:** AFL începe cu o fază de *calibrare* și execută câteva teste deterministice, apoi trece la generarea aleatorie și mutații. În interfața AFL, veți vedea informații precum: *cycles done*, *total execs*, *unique crashes*, *unique hangs*, viteză (exec/sec) etc. Lăsați AFL să ruleze pentru o perioadă (chiar și câteva minute pot fi suficiente să găsească crash-uri simple; pentru bug-uri mai adânci poate fi necesar să ruleze cu orele, dar în laborator vom limita timpul).
4. **Analiza rezultatelor:** Dacă AFL găsește un crash (adică programul a terminat cu o eroare de segmentare sau altă abatere), acesta va salva input-ul respectiv în directorul `outputs/` subfolder **crashes/** (ex: `id:000000_xyz`). Putem deschide acel fișier să vedem ce conținea sau îl putem rula manual cu programul (în afara AFL, eventual sub un debugger) pentru a observa comportamentul. AFL clasifică crash-urile unice după *stack hash* – deci dacă vede mai multe crash-uri cu aceeași cauză, le pune ca unul singur unic.
5. **Exemplu scenariu fuzzing:** Să zicem că fuzzăm un program care procesează string-uri de la intrare și are o vulnerabilitate la un anumit format special de date. AFL va începe de la intrările seed (poate un string gol), apoi va genera variante: va adăuga caractere random, le va schimba, combina ș.a. Dacă la un moment dat generează un input care provoacă un buffer overflow în program, acesta va crăpa. AFL detectează crash-ul și raportează. De exemplu, *unique crashes: 1* va apărea în status, și în `outputs/crashes` vom avea fișierul ce a produs overflow. Putem apoi să examinăm fișierul (de exemplu conține un anumit șir lung) și să reproducem problema.
6. **Întreruperea și reluarea fuzzing-ului:** AFL se poate opri cu `Ctrl+C`. La prima apăsare de Ctrl+C, AFL va *pauza* și va afișa un sumar (câte case a generat etc.) și vă va întreba dacă doriți să ieșiți. Apăsați încă o dată Ctrl+C pentru a confirma ieșirea. Rezultatele rămân în `outputs/`. Dacă doriți să reluați mai târziu fuzzing-ul, puteți folosi conținutul din `outputs/queue` ca input seed (sau folosiți direct `-i outputs/queue` ca seed pentru o sesiune nouă, ceea ce permite continuarea de unde a rămas în mare parte).

Bune practici AFL: începeți cu input seed cât de cât reprezentativ (dacă aveți un format complex, dați un exemplu valid minimal ca seed, altfel AFL va nimeri mai greu căile utile). Lăsați fuzzing-ul să ruleze suficient (în limita timpului de laborator vom face o scurtă demonstrație). De asemenea, puteți rula mai multe instanțe AFL în paralel (pe sisteme multicore) pe același corp de test pentru a acoperi mai rapid – AFL++ suportă *parallel fuzzing* (master/slave instances). În contextul nostru, vom rula doar o instanță.

Prin fuzzing cu AFL se pot descoperi vulnerabilități serioase. AFL a fost folosit în trecut la descoperirea multor bug-uri CVE în aplicații cunoscute, datorită eficienței sale. Este un instrument esențial în arsenalul de testare dinamică pentru aplicații native.

4. Exemple concrete și exerciții practice

În această secțiune vom aplica practic cunoștințele, efectuând **exerciții pas cu pas** cu fiecare instrument. Scopul este ca studenții să ruleze efectiv instrumentele pe aplicații țintă (de laborator) și să interpreteze rezultatele. Durata totală alocată este ~30 minute, așa că fiecare sub-secțiune va fi un exemplu scurt, dar reprezentativ:

- **4.1 Testarea unei aplicații C/C++ cu Valgrind** – Vom rula Valgrind pe un program C/C++ vulnerabil (cu bug-uri de memorie) și vom identifica problemele raportate.
- **4.2 Fuzzing pe un program vulnerabil folosind AFL** – Vom configura și porni AFL pe o aplicație simplă cu vulnerabilitate, observând cum AFL găsește un crash.

Fiecare exemplu va include **pașii de urmat** și așteptările (de ex., "Valgrind ar trebui să raporteze o eroare de…", "AFL va găsi un crash și îl veți vedea în…", etc.). Studenții sunt încurajați să urmărească activ pașii și să pună întrebări în timpul fiecărui exercițiu.

4.1 Testarea unei aplicații C/C++ cu Valgrind

Sarcină: Avem un mic program C, `exemplu.c`, care conține intenționat o problemă de memorie (de exemplu, alocă memorie cu malloc și uită să o elibereze, și accesează un vector cu indice în afara limitei). Vom compila programul și îl vom rula sub Valgrind pentru a detecta problemele.

Pasul 1 – Compilarea programului:
Salvați codul de mai jos ca `exemplu.c`:

```
c
Copy
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(int argc, char *argv[]) {
    char *buffer = malloc(10);
    strcpy(buffer, "Salut lume!");  // copiem 11 caractere într-un buffer de
10 -> overflow
    if(buffer[0] == 'S') {
        printf("Mesaj: %s\n", buffer);
    }
    // intenționat nu facem free(buffer)
    return 0;
}
```

Compilați cu: `gcc -g -o exemplu exemplu.c`. Flagul `-g` este important pentru ca Valgrind să poată indica liniile din cod.

Pasul 2 – Rularea cu Valgrind:

Executați: `valgrind --leak-check=full ./exemplu`. Programul nostru *ar rula aparent normal* (poate afișa „Mesaj: Salut lume!”), dar Valgrind va intercepta problemele:

- Va raporta o eroare de **Invalid write** sau **Invalid read** din cauza suprascrierii buffer-ului (`strcpy` copiază 11 caractere în 10 bytes alocați).
- La final, în sumarul de memorie, va raporta **definitely lost: 10 bytes in 1 blocks** (leak-ul de 10 bytes pentru care nu am apelat free).

Pasul 3 – Interpretarea rezultatului:

Uitați-vă în output-ul Valgrind (în terminal). Ar trebui să vedeți ceva de genul (parțial):

```
yaml
Copy
==XXXXX== Invalid write of size 1
==XXXXX==    at 0xXXXXXXXX: strcpy (vg_replace_strmem.c:xxx)
==XXXXX==    by 0xXXXXXXXX: main (exemplu.c:8)
==XXXXX==  Address 0xYYY is 0 bytes after a block of size 10 alloc'd
==XXXXX==    at 0xXXXXXXXX: malloc ...
==XXXXX==    by 0xXXXXXXXX: main (exemplu.c:7)
...
==XXXXX== LEAK SUMMARY:
==XXXXX==    definitely lost: 10 bytes in 1 blocks
==XXXXX== ...
```

Acest output indică clar cele două probleme:

- **Invalid write** – la linia 8 în exemplu.c (corespunzător lui `strcpy`), adresa accesată este imediat după blocul de 10 bytes (deci am scris în afara zonei alocate).
- **Leak** – 10 bytes neeliberați (alocați la linia 7, malloc).

Pasul 4 – Remediere și reverificare (opțional):
Pentru a vedea cum ajută Valgrind la fixarea bug-urilor, modificați codul: schimbați alocarea la dozen `malloc(12)` să aibă destul spațiu (sau folosiți `strncpy` limitând la 9 caractere + terminator) și adăugați `free(buffer)` înainte de `return`. Recompilați și rulați din nou sub Valgrind. De această dată, **Valgrind ar trebui să raporteze "ERROR SUMMARY: 0 errors" și în LEAK SUMMARY tot 0 bytes pierduți**. Astfel confirmăm că problemele au fost rezolvate.

Rezultat așteptat: Studenții vor vedea practic cum Valgrind scoate la iveală erori de memorie. Acest exercițiu întărește importanța folosirii uneltelor automate pentru a detecta vulnerabilități de memorie ce altfel ar putea conduce la exploatări (buffer overflow-ul detectat ar fi o vulnerabilitate). În plus, evidențiază procesul de *debug-fix-retest*: găsești problema cu Valgrind, apoi o fixezi și confirmi cu Valgrind că nu mai există.

4.2 Fuzzing unui program vulnerabil cu AFL

Sarcină: Vom utiliza AFL pentru a descoperi o vulnerabilitate de tip crash într-un program. Programul țintă, `vuln.c`, citește dintr-un fișier de intrare un șir și, dacă detectează un anumit caracter de control, produce un comportament anormal (ex: dereferențiere de NULL). Scopul este ca AFL, fără cunoaștere prealabilă, să găsească acel input problematic care cauzează crash.

Pasul 1 – Pregătirea programului vulnerabil:
Scrieți următorul cod simplu în `vuln.c`:

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

int main(int argc, char* argv[]) {
    FILE *f = fopen(argv[1], "r");
    if(!f) return 1;
    char buf[100];
    if(fgets(buf, sizeof(buf), f)) {
        // daca gasim caracterul '!' la inceput, simulam o problema
```

```
        if(buf[0] == '!') {
            char *p = NULL;
            *p = 'X';  // crash: dereferentiem pointer NULL
        }
    }
    fclose(f);
    return 0;
}
```

Acest program citește un fișier (specificat ca argument) și dacă primul caracter este `!`, forțează un segfault (dereferențiind un pointer NULL). În mod normal, programul nu face nimic vizibil, doar termină. Dar noi vrem ca fuzzing-ul să-l facă să ajungă la acea condiție.

Pasul 2 – Compilarea cu AFL instrumentation:

Asigură-te că AFL (sau AFL++) este instalat. Compilează folosind wrapper-ul AFL:

```
bash
afl-gcc -g -o vuln vuln.c   # sau afl-clang-fast, depinde de instalare
```

Acum avem binarul `vuln` instrumentat pentru AFL.

Pasul 3 – Directorul de input-uri inițiale:

Cream un folder `testcases/` și în el punem un fișier de pornire. Chiar și un fișier gol poate fi bun. Să facem totuși unul minimal:

```
bash
mkdir testcases
echo "hello" > testcases/input.txt
```

Avem așadar un fișier de 5 bytes care **nu** declanșează bug-ul (nu începe cu `!`).

Pasul 4 – Lansarea AFL:

Executăm fuzzing-ul:

```
bash
afl-fuzz -i testcases -o findings -- ./vuln @@
```

AFL va verifica parametrii și va începe rularea. Veți vedea interfața text: timing, cycles, etc. La început, `unique crashes` = 0.

Lăsăm AFL să ruleze. Acest bug e foarte ușor de găsit, așa că în câteva secunde ar trebui să se schimbe ceva: Uitați-vă la linia "unique crashes" – dacă devine `1`, înseamnă că AFL a găsit input care cauzează segfault.

Pasul 5 – Observarea rezultatului:
Presupunând că AFL a găsit crash-ul, apăsați `Ctrl+C` o dată pentru a pune pauză și vedea sumarul. Ar trebui să raporteze ceva gen "Found 1 crashes". Apăsați din nou Ctrl+C să ieșiți complet.

Acum, verificați directorul `findings/crashes/`. Ar trebui să existe un fișier, de exemplu `id:000000,sig:11,src:000000,time:...,+sig11` (numele conține informații, `sig:11` indică semnalul SIGSEGV). Deschideți acel fișier (e text). Cel mai probabil, conține pur și simplu caracterul `!` ca prim caracter (și posibil restul "hello" de la seed sau alte caractere).

Felicitări, ați obținut cu AFL un **caz de test minimizat** care reproduce vulnerabilitatea! Practic, AFL a încercat să modifice primul byte al fișierului de intrare în diferite variante până a dat peste `!` care a declanșat segfault-ul.

Pasul 6 – Reproducerea manuală (pentru confirmare):
Rulați programul manual cu acel fișier: `./vuln findings/crashes/id:000000*` (tasta Tab pentru autocompletare a numelui). Ar trebui ca programul să se oprească cu *segmentation fault*. Ați confirmat astfel că input-ul descoperit de AFL este valid și produce crash.

Pasul 7 – Curățenie și reflecție:
Exercițiul ne-a arătat că AFL poate găsi chiar și bug-uri foarte ascunse, dar și bug-uri triviale ca în exemplul nostru. Important e procesul: instrumentare, definire input minim, lăsat fuzzer-ul să ruleze.

Rezultat așteptat: Studenții vor vedea AFL în acțiune, afișând statistici live și reușind să găsească un crash. Vor înțelege ce se întâmplă în spate (AFL modifică fișierul, încearcă diferite caractere la început și nimerește `!`). De asemenea, vor învăța să recupereze output-ul (fișierul de crash) și să-l folosească pentru a reproduce și eventual a depana problema. Chiar dacă exemplul e simplu, el replică un flux real de fuzzing: pregătire, rulare, obținere crash, apoi investigație asupra crash-ului.

5. Interpretarea rezultatelor și remedierea vulnerabilităților

După efectuarea testelor dinamice, ne vom concentra pe **analiza rezultatelor obținute** și pe **măsurile de remediere**. Este esențial ca studenții să înțeleagă nu doar cum să găsească vulnerabilități, ci și cum să le interpreteze (prioritizeze) și cum ar putea fi rezolvate în cod.

Identificarea vulnerabilităților în rapoarte:

- În urma rulării Valgrind, vulnerabilitățile apar sub formă de **erori de memorie**. Trebuie să interpretăm fiecare mesaj: de exemplu, "Invalid read/write" indică acces ilegal (posibil buffer overflow sau pointer nevalid), "Conditional jump or move depends on uninitialized value" indică utilizarea unei variabile neinitialize (poate duce la comportament imprevizibil), iar secțiunea *LEAK SUMMARY* evidențiază scurgerile de memorie. Fiecare element raportat corespunde unei potențiale vulnerabilități de securitate sau cel puțin unei erori de programare severă. Programatorul ar trebui să se uite la liniile de cod menționate și să înțeleagă cauzele (ex: de ce buffer-ul a fost depășit? unde trebuia eliberată memoria și nu s-a eliberat?).
- Rularea AFL produce output mai sumar: ne interesează în special fișierele din `crashes/`. Fiecare fișier reprezintă un **caz de test care determină un crash**. Interpretarea vulnerabilității se face analizând acel input și codul programului. De exemplu, dacă fișierul conține un text foarte lung care a dus la buffer overflow, vulnerabilitatea ar fi lipsa verificării lungimii inputului. Dacă inputul conține caractere speciale care nu au fost anticipate (ex: un caracter Unicode ce a stricat logica), atunci vulnerabilitatea poate fi un filtru incomplet sau o conversie greșită. Un aspect important la interpretare este și **prioritizarea**: AFL poate găsi zeci de crash-uri, dar unele pot fi variații ale aceleiași probleme fundamentale. Se analizează stack trace-urile (dacă rulăm programul cu AddressSanitizer sau sub debugger cu acel input) pentru a identifica exact tipul și locația bug-ului.

Exemple de fixare a problemelor:

- Pentru bug-urile de memorie (Valgrind/AFL): de regulă remedierea implică corecții de cod în limbajele native:
    - Buffer overflow-ul poate fi rezolvat alocând corect dimensiunile necesare sau introducând verificări (e.g., folosirea `snprintf` în loc de `strcpy`, verificarea indicilor la accesul array-urilor).

- *Use-after-free* – necesită regândirea logicii de alocare/eliberare: asigurați-vă că nu se mai folosește pointerul după free sau implementați o mecanism de *smart pointer*.
- Memory leak – adăugați `free()` corespunzătoare (eventual utilizând un pattern RAII sau destructor în C++ pentru a automatiza).
- Erori de conversie numerică – de ex, dacă fuzzing-ul a descoperit că un număr prea mare cauzează overflow intern, se poate adăuga o limitare la valoarea maximă acceptată sau se folosește un tip de date mai mare.

Recomandări pentru prevenirea vulnerabilităților în cod (pe viitor):

- Adoptarea unor **bune practici de programare**: de exemplu, pentru C/C++ – utilizarea *safe libraries* sau funcții cu limite (fgets în loc de gets, strncpy vs strcpy etc.), testarea riguroasă a oricărei operații pe memorie. Pentru aplicații web – principiul *never trust user input*: toate datele de la client să fie validate sau escăpate corespunzător.
- Integrarea unor instrumente **automatizate încă din faza de dezvoltare**: analiza statică (SAST) poate prinde din timp unele greșeli (ex: un tool ca SonarQube ar semnala folosirea lui gets() sau concatenarea de stringuri în SQL). Iar rularea regulată a testelor dinamice (DAST) pe build-urile de staging poate prinde vulnerabilități înainte de producție.
- **Patch management**: uneori vulnerabilitățile provin și din folosirea unor versiuni vechi ale unor biblioteci (ex: o versiune de OpenSSL sau jQuery cu bug cunoscut). Un scanner dinamic ar putea raporta asta ("componentă cu vulnerabilitate cunoscută"). Rezolvarea e actualizarea la versiuni sigure.
- Scrierea de **teste unitare/integrare de securitate**: de exemplu, puteți scrie un test care să ruleze o suită de input-uri (poate generate cu Radamsa) printr-o funcție critică și să asigure că nu se prăbușește.

Prin interpretarea atentă a rezultatelor și aplicarea remediilor potrivite, scopul final este ca, dacă rulăm din nou instrumentele de analiză dinamică pe aplicația fixată, acestea să nu mai raporteze vulnerabilitățile respective. De pildă, după fixare, Valgrind arată 0 erori, ZAP nu mai găsește SQLi etc. – acesta este indicatorul că aplicația a devenit semnificativ mai sigură.

6. Concluzii și bune practici

În concluzie, laboratorul de analiză dinamică ne-a familiarizat cu instrumente și tehnici esențiale pentru testarea securității aplicațiilor software. Recapitulăm câteva idei cheie, evidențiem limitările DAST și sugerăm cum să integrăm aceste instrumente în procesul de dezvoltare, alături de alte metode de asigurare a securității:

- **Limitările DAST:** Analiza dinamică, deși valoroasă, nu este un glonț de argint. Un aspect este acoperirea incompletă: instrumentele DAST testează aplicația doar în limitele scenariilor de input care le sunt furnizate sau generate. Dacă există funcționalități care nu au fost exercitate în timpul testelor, vulnerabilitățile din acele zone pot trece neobservate. De exemplu, un fuzzer poate să nu nimerească un input exact care declanșează un bug foarte specific, sau un scanner web poate să nu autentifice un anumit rol de utilizator și astfel să rateze probleme din acea zonă. De asemenea, DAST poate produce **false negative** – să nu raporteze o vulnerabilitate existentă (poate pentru că nu a detectat cum să o declanșeze). În plus, uneori apar și **false positive** – mai rar la fuzzing (un crash e un crash real), dar la scanere web unele alerte pot fi zgomot (de exemplu, avertismente informative care nu sunt exploatabile). Un alt aspect: testarea dinamică a aplicațiilor poate fi consumatoare de timp și resurse (un fuzz serios poate rula zile, o scanare intensă pe un site mare poate dura ore și poate încărca serverul). Trebuie calibrat efortul DAST cu nevoile concrete (de exemplu, să rulați fuzzing targetat pe componente critice, nu orbește pe tot).
- **Integrarea analizei dinamice în SDLC:** (*Software Development Life Cycle*) – Ideal, securitatea trebuie înglobată în fiecare etapă a dezvoltării:
    - În faza de implementare, pe lângă code review și analiza statică, dezvoltatorii pot scrie **teste de securitate** și pot rula local instrumente precum Valgrind (pentru componente C/C++). De exemplu, înainte de commit, dacă e cod C, să ruleze o suită de teste sub Valgrind și să se asigure că nu apar erori.
    - În faza de testare și QA, se pot include **scanări automate regulate**: integrând OWASP ZAP în pipeline-ul de CI (ZAP are mod headless și API care permit rularea automată a scanării după fiecare build și generarea de rapoarte). Similar, se pot rula campanii de fuzzing (poate nu la fiecare commit, dar la milestone-uri importante) – de exemplu, înainte de o versiune majoră a unei biblioteci, rulezi AFL pe funcțiile sale publice câteva ore.
    - În faza de stagiu/pre-producție, se poate realiza un **test de penetrare** mai amplu (care combină DAST manual și automat) folosind Burp Suite Pro sau alte

instrumente, eventual cu echipe specializate, pentru a descoperi vulnerabilități pe care automatizările standard nu le-au prins.

- După deploy, DAST poate continua sub forma monitorizării: de exemplu, folosirea unui scanner precum Nikto sau OpenVAS periodic pe aplicația live (cu acordul echipei, desigur) pentru a detecta noi probleme (poate introduse de update-uri de infrastructură).

Bune practici generale DAST:

- Automatizați pe cât posibil, dar **nu excludeți componenta umană**: un pen-tester cu experiență poate gândi scenarii pe care scanerul nu le încearcă.
- **Nu testați pe orbește sisteme în producție** (fuzzing-ul și scanarea pot provoca încărcare mare sau chiar downtime dacă aplicația e sensibilă). Faceți aceste teste în medii controlate sau cu acord explicit și în afara orelor de vârf.
- **Documentați rezultatele**: fiecare vulnerabilitate identificată ar trebui consemnată, cu pași de reproducere și sugestii de fix, astfel încât dezvoltatorii să poată acționa. Rulați din nou testele după fixare pentru a verifica.
- Țineți la curent uneltele: actualizați Burp/ZAP la ultimele versiuni, actualizați dicionarele de payload-uri etc., deoarece apar mereu semnături noi pentru atacuri noi.
- Încercați să **gândiți ca un atacator**: DAST eficient înseamnă să nu vă rezumați doar la a apăsa butoane, ci a înțelege ce se întâmplă. De exemplu, dacă fuzzing-ul găsește un crash, gândiți-vă "Cum ar putea fi exploatat asta? Ce ar însemna pentru securitate?". Dacă scannerul găsește un XSS, imaginați-vă ce daune ar putea face – astfel veți putea comunica echipei impactul (și îi veți motiva să remedieze).