Lucrare practica / Lucrare de laborator nr. 2

Tema: Utilizarea instrumentelor de analiză dinamică pentru evaluarea vulnerabilităților produselor program

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.

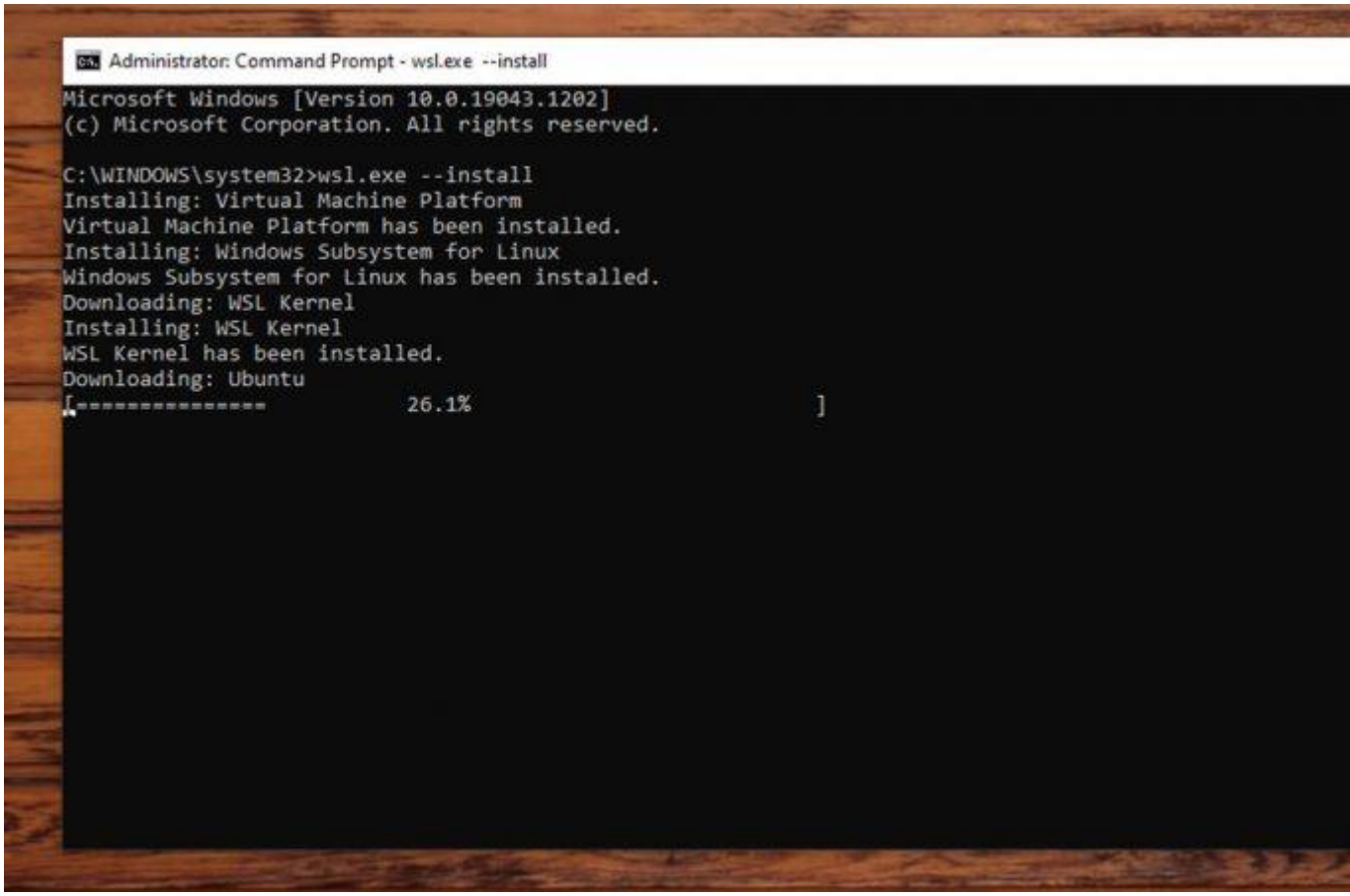
- Sistem de operare: **Linux** este recomandat pentru instrumente ca Valgrind și AFL (care au suport nativ excelent pe Linux). **Windows** va fi folosit mai ales pentru instrumente de testare web (Burp, ZAP) – acestea sunt multi-platformă, deci pot rula și pe Linux, însă în practică multe testări web se fac de pe stații Windows. În cazul în care lucrați pe Windows și trebuie să folosiți unelte ca Valgrind/AFL care sunt native Linux, **puteți folosi o mașină virtuală Linux sau subsistemul Windows pentru Linux (WSL).**

Configurarea și utilizarea WSL (Windows Subsystem for Linux)

Instalarea WSL pe Windows 11: Windows 11 suportă nativ WSL 2, permițând rularea unui kernel Linux real în Windows. Pentru a instala, deschide **Windows PowerShell** sau **Windows Terminal** cu drepturi de administrator și rulează comanda:

```
powershell
wsl --install
```

Aceasta va activa componentele necesare (Platforma Mașină Virtuală și Subsistemul Windows pentru Linux), va descărca kernel-ul Linux și va instala implicit distribuția Ubuntu. De obicei, instalarea decurge automat, necesitând repornirea sistemului la final. După repornire, WSL va continua configurarea distribuției Linux (Ubuntu implicit) și îți va cere să creezi un **utilizator Unix și o parolă** pentru noul mediu Linux. Dacă dorești altă distribuție (de exemplu Debian), poți specifica în comanda de instalare flagul `-d <Distribuție>` (ex: `wsl --install -d Debian`) . Poți lista distribuțiile disponibile cu `wsl --list --online` pentru a alege pe cea dorită.



Execuția comenzii `wsl --install` în promptul de comandă Windows 11. WSL activează funcțiile necesare (Virtual Machine Platform, Windows Subsystem for Linux) și descarcă automat distribuția Linux selectată (implicit Ubuntu).

Configurarea mediului Linux în WSL: După instalare, poți lansa distribuția Linux (ex. Ubuntu) din meniul Start sau executând `wsl` în terminal. Vei avea un prompt Bash în care poți rula comenzi Linux. Este recomandat să actualizezi pachetele imediat:

```
bash
sudo apt update && sudo apt upgrade -y
```

Acest mediu WSL îți permite să instalezi utilitare Linux (gcc, make, Valgrind, etc.) folosind managerul de pachete al distribuției (apt pentru Ubuntu/Debian). De exemplu, pentru a instala utilitarul de dezvoltare **build-essential** (compilator C/C++ și utilitare make) rulează `sudo apt install build-essential`. Fișierele de pe discul Windows pot fi accesate din WSL sub `/mnt/c/` (unitatea C:), astfel încât poți compila și rula în WSL cod sursă aflat pe Windows. WSL 2 oferă suport și pentru aplicații Linux cu interfață grafică (folosind serverul X integrat în

Windows 11), însă pentru scopul nostru (unelte de securitate linie de comandă și servere web) nu vom avea nevoie de interfață grafică Linux.

Notă: Asigură-te că funcția de **virtualizare hardware** este activată în BIOS/UEFI (VT-x pentru Intel sau AMD-V pentru AMD), deoarece WSL 2 rulează un nucleu Linux într-o mașină virtuală ușoară. Pe majoritatea PC-urilor moderne această opțiune este activată implicit, dar dacă întâmpini erori la pornirea WSL, verifică setările de virtualizare ale sistemului.

Instalarea alternativelor pentru Windows

În timp ce WSL permite rularea directă a uneltelor Linux în Windows, există și alternative ori soluții complementare pentru a rula aceste instrumente sau echivalentele lor direct în Windows, atunci când este necesar.

Dr. Memory – alternativă la Valgrind pe Windows

De ce Dr. Memory? Valgrind (în special instrumentul Memcheck) nu funcționează direct pe programe Windows native, deoarece necesită un mediu Linux. Ca alternativă, pentru a depista erori de memorie în aplicații Windows (executabile .exe), se folosește **Dr. Memory**, un debugger de memorie open-source inspirat de Valgrind, compatibil cu Windows. Dr. Memory poate detecta accesări ilegale de memorie, citiri din variabile neinițializate, scurgeri de memorie, etc., în programe Windows pe 32-bit sau 64-bit.

Instalarea Dr. Memory: Descărcați pachetul de instalare Windows (fișier `.msi`) de pe site-ul oficial **drmemory.org**. Executați installer-ul și urmați pașii obișnuiți (Next, I Agree, Install). Instalatorul va adăuga Dr. Memory în variabila de mediu PATH a utilizatorului curent pentru a putea fi invocat din linia de comandă. *(Dacă observați că după instalare comanda nu este recunoscută în Command Prompt, e posibil să fie nevoie de o re-logare sau repornire pentru ca actualizarea PATH să aibă efect, conform notelor de pe site-ul oficial.)* Alternativ, există o distribuție zip care poate fi extrasă manual, caz în care trebuie să adăugați manual folderul `DrMemory\bin` în PATH, dar metoda cu .msi este mai simplă.

Utilizarea Dr. Memory: Dr. Memory se folosește din linia de comandă, similat cu Valgrind. Pentru a analiza un program Windows, deschide un **Command Prompt** sau **PowerShell** (nu e necesar admin) și rulează comanda:

```
bat
drmemory.exe -- path\la\program.exe [argumente]
```

Introducând `drmemory.exe --` înaintea comenzii de lansare a programului, programul respectiv va rula sub monitorizarea Dr. Memory. De exemplu, dacă ai un executabil `test.exe` compilat cu debug symbols, poți rula:

```
bat
drmemory.exe -- C:\Proiecte\build\test.exe input1.txt
```

Dr. Memory va porni programul și va intercepta erorile de memorie. La final (sau la momentul producerii unui crash) va afișa un raport fie în consolă, fie într-un fișier de log (de exemplu `results.txt`). Mesajele Dr. Memory arată similar cu cele Valgrind, indicând tipul erorii și stiva de apeluri. Un exemplu de sumar de ieșire Dr. Memory ar putea fi:

```
scss
~~Dr.M~~ ERROR(s) FOUND: 5 unique, 5 total, 574 byte(s) of leak(s)
~~Dr.M~~ 0 unique, 0 total, 0 byte(s) of uninitialized access(es)
```

(semnificând că au fost găsite 5 scurgeri de memorie totalizând 574 bytes și nicio utilizare de memorie neinițializată). Pentru a interpreta detaliat aceste rezultate și a identifica locațiile în cod, parcurge raportul generat, care listează adresele și eventual liniile de cod asociate fiecărei probleme. Dr. Memory se poate integra și în Visual Studio ca un instrument extern (instalatorul adaugă o scurtătură în meniul Tools dacă detectează Visual Studio), facilitând rularea directă pe proiectele C/C++ din IDE.

VirtualBox – mașină virtuală Linux ca alternativă la WSL

De ce un VM Linux? În majoritatea cazurilor, WSL 2 este suficient pentru a rula unelte Linux pe Windows. Totuși, dacă unealta necesară are cerințe speciale de kernel sau de mediu (de exemplu rularea unor drivere, module kernel, sau dacă doriți să simulați un mediu de atac separat), puteți folosi o mașină virtuală Linux prin Oracle **VirtualBox** (sau Hyper-V, VMWare etc.). VirtualBox permite rularea unui sistem de operare complet separat, ceea ce oferă izolare mai bună și suport deplin pentru toate funcționalitățile Linux, cu prețul unui consum de resurse mai mare.

Instalarea VirtualBox: Descărcați VirtualBox pentru Windows de pe site-ul oficial Oracle (secțiunea *Downloads* pentru Windows hosts). Instalați-l rulând executabilul și urmați pașii din wizard (puteți lăsa setările implicite). După instalare, descărcați o imagine ISO a unei distribuții Linux (de ex. Ubuntu LTS de pe ubuntu.com). Porniți aplicația VirtualBox și creați o mașină virtuală nouă (**New**): alegeți un nume (ex. "UbuntuVM"), tip **Linux**, versiune Ubuntu 64-bit. Alocați memorie RAM (de ex. 2048 MB sau mai mult, în funcție de disponibil) și spațiu pe disc (creați un disk virtual de ex. 20 GB sau mai mult). La pasul de selectare a imaginii de instalare, indicați ISO-ul Ubuntu descărcat. După finalizare, porniți VM-ul (Start) și urmați procesul de instalare Ubuntu ca pe un PC obișnuit. După instalare, veți avea un mediu Linux complet desktop în care puteți instala Valgrind, AFL, etc., ca pe orice alt sistem Linux.

Avantaje și dezavantaje: VirtualBox oferă un mediu 100% compatibil, deci dacă întâmpinați probleme cu unele unelte în WSL, VM-ul Linux este soluția. De exemplu, unealta de fuzzing

AFL poate necesita acces raw la interfața de sistem de fișiere sau disable ASLR global – lucruri posibile într-un VM dedicat fără a afecta sistemul gazdă. Totodată, într-un VM puteți rula și servicii de rețea cu interfață grafică (ex. aplicații web vulnerabile pentru testare) separat de rețeaua principală. Dezavantajul este consumul de resurse: va rula un întreg sistem de operare în paralel cu Windows, deci asigurați-vă că aveți suficientă memorie RAM și CPU. De asemenea, fișierele pot fi transferate prin foldere share sau rețea virtuală între Windows și VM. Pentru majoritatea scenariilor din acest ghid, WSL este suficient, dar țineți cont de VirtualBox ca opțiune de rezervă.

(Notă: dacă folosiți Hyper-V sau alte soluții, procesul e similar. Asigurați-vă însă că Hyper-V este dezactivat dacă folosiți VirtualBox, deoarece pot exista conflicte.)

3. Instalarea și utilizarea uneltelor de securitate

În continuare vom detalia modul de instalare (atât pe WSL/Linux cât și alternative Windows dacă există) și utilizare pentru fiecare unealtă: **Valgrind** și **AFL.**

3.1 Valgrind (Memcheck) – detectarea erorilor de memorie

Instalare (Linux/WSL): Valgrind se instalează ușor pe distribuțiile Linux uzuale prin managerul de pachete. De exemplu, pe Ubuntu/Debian (inclusiv în WSL) rulează:

```
bash
sudo apt install valgrind
```

Verifică instalarea cu `valgrind --version` (trebuie să afișeze versiunea Valgrind instalată). Valgrind nu are interfață grafică; este o suită de instrumente linie de comandă, cea mai folosită fiind **Memcheck** (implicită), care detectează probleme de memorie. Nu există Valgrind nativ pentru executabile Windows, de aceea pe Windows recomandăm Dr. Memory (discutat mai sus) pentru aplicațiile Windows. În acest ghid vom folosi Valgrind în mediul WSL/Ubuntu pentru a analiza aplicații compilate acolo (sau cross-compile din Windows cu GCC în WSL).

Pregătirea programului de test: Pentru a beneficia la maxim de mesajele Valgrind, compilă aplicațiile C/C++ **cu simboluri de depanare** (`-g`) și ideal fără optimizări excesive (`-O0` sau -O1) . Astfel, Valgrind va putea indica numerele de linie în cod unde apar erori. Să presupunem un program simplu `test.c` cu bug-uri de memorie:

```
c
```

```
#include <stdlib.h>
#include <string.h>
int main() {
    char *data = malloc(10);
    strcpy(data, "Salut lume!");  // Depășește alocarea de 10 bytes
    // (bug: nu eliberăm memoria cu free)
    return 0;
}
```

Acest program are două probleme: copiere peste limita buffer-ului alocat (scriere în afara zonei de 10 octeți) și *memory leak* (nu se apelează `free`). Îl compilăm cu debug info:

```
bash
gcc -g -O0 -o test test.c
```

Rularea programului cu Valgrind: Pentru a analiza, rulăm:

```
bash
valgrind --leak-check=yes ./test
```

Flagul `--leak-check=yes` instruiește Memcheck să raporteze detaliat scurgerile de memorie la terminare. La rulare, programul se va executa mult mai lent (Valgrind este un emulator, încetinește execuția de ~20x), iar la final vom vedea mesajele de eroare. De exemplu, ieșirea Valgrind pentru programul de mai sus ar arăta (prescurtat):

```
yaml
==12345== Invalid write of size 1
==12345==    at 0x... : strcpy (in /usr/lib/.../libc.so)
==12345==    by 0x... : main (test.c:5)
==12345==  Address 0x... is 0 bytes after a block of size 10 alloc'd
==12345==    at 0x... : malloc (vg_replace_malloc.c:...)
==12345==    by 0x... : main (test.c:4)
==12345==
==12345== HEAP SUMMARY:
==12345==    in use at exit: 10 bytes in 1 blocks
==12345==    definitely lost: 10 bytes in 1 blocks
==12345==    indirectly lost: 0 bytes in 0 blocks
==12345==      possibly lost: 0 bytes in 0 blocks
==12345==    still reachable: 0 bytes in 0 blocks
==12345==         suppressed: 0 bytes in 0 blocks
==12345==
==12345== LEAK SUMMARY:
```

```
==12345==    definitely lost: 10 bytes in 1 blocks
==12345==    possibly lost: 0 bytes in 0 blocks
==12345==    still reachable: 0 bytes in 0 blocks
==12345==         suppressed: 0 bytes in 0 blocks
==12345==
==12345== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Să interpretăm acest raport:

- **„Invalid write of size 1”** – Valgrind a detectat o scriere ilegală de 1 octet. Ne indică și unde: în `strcpy` apelată de `main` la linia 5 din `test.c`. De asemenea, spune că adresa accesată este imediat după un bloc de 10 octeți alocați la linia 4 (prin malloc). Acesta corespunde depășirii buffer-ului (`"Salut lume!"` are 11 caractere cu tot cu terminatorul `\0`, deci depășește cei 10 alocați).
- **„definitely lost: 10 bytes in 1 blocks”** – Rezumatul de la final indică o scurgere de memorie: 10 bytes cu siguranță pierduți (alocați și neeliberați). Asta confirmă că nu am eliberat memoria alocată în `data`. „Definitely lost” în Valgrind înseamnă memorie pierdută sigur – trebuie **remediată**
- **ERROR SUMMARY: 2 errors** – Avem 2 erori distincte (suprascrierea buffer-ului și leak-ul).

În concluzie, Valgrind ne-a evidențiat clar bug-urile. Pentru a **remedia** exemplul, am face alocarea mai mare sau am folosi o funcție sigură (`strncpy`) pentru a evita depășirea, și am adăuga `free(data)` înainte de return. După modificare, rulând din nou Valgrind, mesajele de eroare ar trebui să dispară (EROR SUMMARY: 0 errors).

Valgrind detectează multe tipuri de erori de memorie (buffer overflow pe heap, utilizarea memoriei după free – *use after free*, citirea unei variabile neinițializate, double free, etc.). Folosiți Valgrind în mod iterativ în timpul dezvoltării C/C++ pentru a prinde aceste probleme înainte de a ajunge în producție. Raportările Valgrind includ adesea stiva de apel care a dus la eroare, facilitând localizarea și corecția în cod.

Sfaturi:

- Dacă aveți o aplicație mare, rulați testele unitare sub Valgrind ca să acoperiți cât mai mult cod.
- Ignorați erorile „benigne” numai dacă sunteți sigur (Valgrind poate da uneori „false positives” pe cod optimizat sau biblioteci, dar rar).

- Folosiți opțiuni precum `--track-origins=yes` pentru a afla unde s-a alocat o variabilă neinițializată, sau `--leak-check=full --show-leak-kinds=all` pentru detalii despre leaks.

3.2 AFL (American Fuzzy Lop) – fuzzing automatizat pentru aplicații

Despre AFL: AFL este un fuzzer orientat pe securitate care folosește instrumentare la nivel de cod pentru a genera date de intrare malformate ce explorează căi de execuție diferite în program. Cu alte cuvinte, AFL rulează în buclă programul de test cu intrări mutate (fuzz) și monitorizează acoperirea de cod, încercând să găsească crash-uri sau blocări. AFL a fost inițial dezvoltat de Michał Zalewski și este cunoscut pentru eficiența sa în a descoperi vulnerabilități de tip buffer overflow, use-after-free etc. într-un mod automat.

Instalare (Linux): Pe WSL/Ubuntu puteți instala AFL original sau varianta mai nouă AFL++ (care include optimizări și suport extins). Instalarea se face prin apt:

```
bash
Copy
sudo apt install afl++
```

(Dacă pachetul `afl++` nu e disponibil, încercați `sudo apt install afl` pentru versiunea clasică.) Puteți verifica cu `afl-fuzz --version`. Alternativ, puteți compila din surse de pe GitHub (afl++). În Windows, AFL nu rulează nativ pe executabile Windows (decât eventual sub Cygwin, dar nu e uzual), așa că vom folosi AFL în mediul Linux (WSL sau VM) pentru a fuzz-ui aplicații Linux.

Pregătirea programului pentru fuzzing: AFL funcționează cel mai bine când are acces la instrumentare de cod sursă. Asta înseamnă că trebuie să **compilați programul țintă cu AFL**. AFL oferă un wrapper pentru GCC/Clang (`afl-gcc`, `afl-clang-fast` etc.) care inserează cod de instrumentare. De exemplu, dacă aveți un program C `vuln.c`, îl compilați cu:

```
bash
afl-gcc -g -O0 -o vuln_prog vuln.c
```

Acest executabil rezultat va conține instrumentațiile necesare pentru ca AFL să monitorizeze execuția (practic marchează tranzițiile în graful de execuție). Asigurați-vă că dezactivați protecții precum address sanitizer sau fortify temporar, deoarece AFL vrea să detecteze el crash-urile

direct. De asemenea, pe sistemele moderne, e util să dezactivați *address space layout randomization* (ASLR) pentru reproducibilitate:

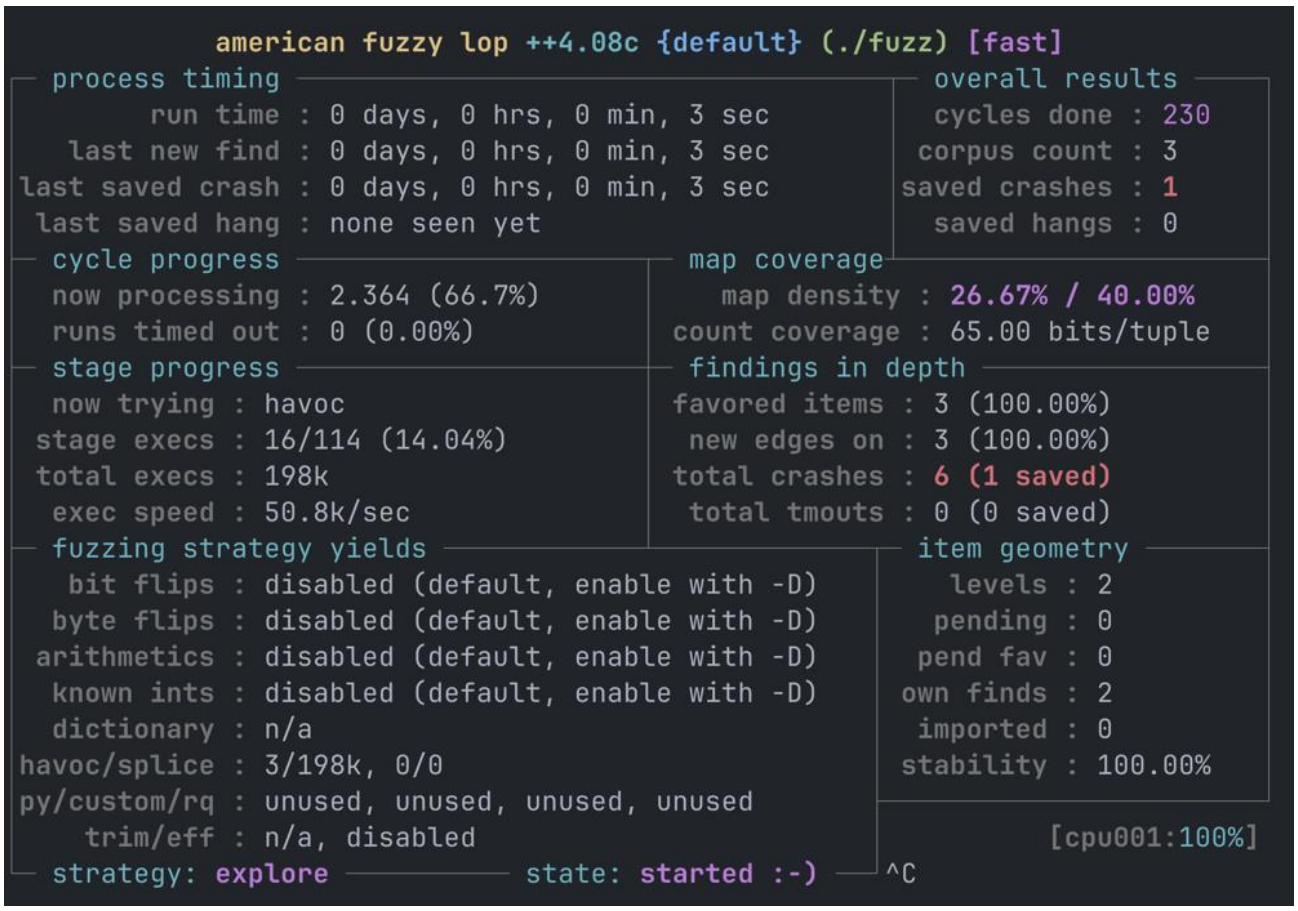
```
bash
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

(sau puteți rula `afl-system-config`, script oferit de AFL++, care aplică setări de performanță și dezactivează mitigările de securitate ce pot împiedica fuzzing-ul eficient).

Executarea AFL: AFL are nevoie de: 1) un folder cu fișiere de intrare inițiale (*seed inputs*), 2) un folder unde să salveze rezultatele (intrări generate, crash-uri etc.), și 3) comanda de rulare a programului instrumentat, cu un marcaj special `@@` în locul numelui fișierului de intrare (dacă programul citește din fișier). De exemplu, dacă vrem să fuzz-uim programul nostru `vuln_prog` care citește date dintr-un fișier, creăm mai întâi un director `inputs/` și punem acolo cel puțin un fișier valid minimal (seed – exemplu: `test1.txt`). Apoi rulăm AFL astfel:

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

Parametrul `-i` specifică folderul cu teste inițiale, `-o` folderul unde va pune rezultatele. `--` marchează finalul opțiunilor AFL și ce urmează e comanda pentru programul țintă. `@@` va fi înlocuit automat de AFL cu calea fiecărui fișier de test pe rând. Dacă programul țintă citește de la **stdin** în loc de fișier, atunci nu folosiți `@@` – AFL va trimite automat conținutul fișierelor seed pe intrarea standard a programului (în acest caz comanda ar fi `afl-fuzz -i inputs -o outputs -- ./vuln_prog`, fără `@@`). AFL va porni și va afișa un **ecran de status** în consolă, actualizat în timp real.



Ecranul de stare al AFL++ în timpul unei sesiuni de fuzzing. AFL afișează statistici precum timpul de rulare, numărul de execuții pe secundă, numărul de intrări în coadă, noile trasee de cod descoperite, precum și ***numărul de crash-uri sau blocări (hangs) descoperite****. De exemplu, în partea dreaptă sus vedem „saved crashes: 1” indicând că o intrare care cauzează un crash a fost găsită și salvată.*

În timpul execuției, AFL va genera numeroase variații ale inputurilor (folosind mutatori genetici: bit flips, adăugare de pattern-uri, valori interesante etc.) și va detecta automat dacă vreun input cauzează crash (segmentation fault, abort, etc.) sau face programul să se blocheze (hang - timp de execuție depășit). Fiecare caz de crash unic (după semnătură) este salvat în directorul `outputs/crashes/` cu un nume descriptiv (de ex. `id:000000,sig:11,src:000001,...` unde `sig:11` indică un SIGSEGV). Similar, blocările sunt salvate în `outputs/hangs/`. Directorul `outputs/queue/` va conține toate intrările de test (inclusiv cele inițiale și cele mutate) care au dus la noi traiectorii de execuție - practic corpus-ul extins de teste. AFL rulează până când îl oprești manual (Ctrl+C) sau poți seta un timp maxim.

Exemplu de utilizare: Să presupunem că avem un mic program care procesează un șir de caractere de la stdin și are o vulnerabilitate de depășire de buffer dacă intrarea este prea lungă.

Folosind AFL cu un seed scurt (de ex. "HELLO"), fuzzer-ul va încerca să extindă și să modifice intrarea. AFL va observa că intrările mai lungi duc la noi părți din cod (poate la un overflow) și eventual va genera o intrare care cauzează un crash (de ex. 1000 de caractere 'A'). În momentul acelui crash, AFL îl salvează și incrementă contorul de crash-uri. Dvs. puteți apoi analiza fișierul respectiv (din `outputs/crashes`) pentru a vedea ce a provocat (conținutul, de exemplu 1000 de 'A') și folosi un debugger pe programul dvs. rulând cu acel input pentru a localiza exact cauza (stack overflow, etc.). Vom discuta interpretarea rezultatelor în secțiunea 5.

Alte opțiuni utile: AFL oferă multe opțiuni: `-t <ms>` pentru a seta un timeout custom per execuție (implicit ~1 sec), `-m <megs>` pentru a seta un memory limit (implicit ~50 MB). De asemenea, dacă nu puteți recompile programul, AFL are moduri alternative: modul QEMU (`-Q`) care poate instrumenta binare pe zbor, sau modul „dumb" fără instrumentare (`-n`), însă eficiența scade. Ideal este să aveți sursa și să folosiți instrumentarea normală. AFL++ vine și cu alte tool-uri: `afl-cmin` (minimizează corpus-ul de intrări la cele distincte ca acoperire), `afl-tmin` (minimizează un anumit fișier care cauzează crash la cea mai mică dimensiune care încă produce crash), etc. De asemenea, suportă fuzzing distribuit pe mai multe core-uri (parametrul `-M`/`-S` pentru master/slave instances).

Prin combinarea AFL cu Valgrind sau ASan (Address Sanitizer) puteți detecta nu doar crash-uri efective, ci și condiții de memorie coruptă care nu duc imediat la crash. Un truc este să instrumentați cu AFL și să legați cu AddressSanitizer, astfel AFL va marca intrările care duc la erori ASan (de obicei, se manifestă ca crash cu abort). Acest setup avansat necesită ceva configurare, dar sporește aria de bug-uri detectabile (de exemplu, out-of-bounds care nu imediat crape programul, tot vor fi prinse de ASan).

4. Exemple de utilizare pe aplicații reale

În această secțiune vom trece prin scenarii de aplicare pentru fiecare unealtă, ilustrând modul de folosire pe cazuri practice și ce tip de probleme putem găsi:

4.1 Detectarea erorilor de memorie cu Valgrind pe un program C/C++

Situație: Avem o aplicație C complexă (sau chiar un modul din aplicația noastră) și suspectăm existența unor scurgeri de memorie sau a unui crash aleator. Putem folosi Valgrind în timpul testării pentru a identifica problemele. Să luăm un exemplu concret: un modul de procesare imagini scris în C care la inputuri specifice crăpa. Vom rula bateria de teste sub Valgrind:

- Pas 1: Compilăm totul cu simboluri de debug (`-g`). În cazul unui proiect mare, e posibil să fie deja compilat in Debug mode pentru testare.
- Pas 2: Rulăm testele unitare: `valgrind --leak-check=yes ./test_suite`.
- Pas 3: Valgrind raportează, să zicem, un *Invalid read* într-o funcție de decodare imagine (indică un buffer overread). Vedem în stack trace că se întâmplă în `decode_pixel()` la o anumită linie. De asemenea, la final, Valgrind arată și ceva memorie "still reachable" sau "possibly lost".
- Pas 4: Corelăm informația: invalid read sugerează că se citește dincolo de limită – verificăm codul `decode_pixel` și constatăm că indexează greșit ultimul element al unui array (off-by-one). Îl corectăm.
- Pas 5: Rulăm din nou Valgrind, eroarea de invalid read dispare. Acum rămân doar mesajele legate de memorie neeliberată: "10,000 bytes in 5 blocks are definitely lost" de exemplu. Inspectând stack trace asociat (Valgrind arată unde s-au alocat acele blocuri), identificăm că funcția de încărcare a imaginii alocă un buffer mare pe care nu-l eliberează dacă apare o eroare de validare. Soluția: adăugăm `free` în ramura de eroare sau folosim smart pointers dacă e C++.
- Pas 6: După remediere, rulăm iar Valgrind – **HEAP SUMMARY: in use at exit: 0 bytes** – curat. Astfel, am folosit Valgrind pentru a curăța complet codul de probleme de memorie.

Rezultat: Programul nu mai pierde memorie și nu mai citește în afara limitelor, deci probabil nu va mai crăpa neașteptat. Valgrind ne-a dat încredere că am rezolvat sursele de instabilitate.

4.2 Fuzzing cu AFL pe un program vulnerabil de test

Situație: Să considerăm un program intenționat vulnerabil (gen un **server TCP** care citește linii de text și are o problemă de depășire de buffer). Putem folosi AFL pentru a descoperi automat intrarea ce provoacă exploatarea.

- Pas 1: Pregătim programul pentru fuzzing. Avem acces la cod (să zicem `vuln_server.c`). În loc să-l rulăm ca server real (network fuzzing e posibil dar mai complex), putem adapta o versiune care citește de la stdin o linie (sau dintr-un fișier) – deci îl transformăm într-un utilitar de linie de comandă `vuln_test` ca țintă pentru AFL. Compilăm cu `afl-gcc` instrumentat.
- Pas 2: Creăm un fișier seed `input/seed.txt` cu conținut banal ("ABC").

- Pas 3: Rulăm `afl-fuzz -i input -o findings -- ./vuln_test @@`. AFL începe fuzzing-ul. La început vedem "No crashes yet". După câteva secunde sau minute, apare în status `unique crashes : 1` și se salvează un fișier în `findings/crashes/id:000000,...`. Să zicem că intrarea fatală este `AAAA....A` (500 de 'A').
- Pas 4: Confirmăm: rulăm manual `./vuln_test crashes/id:000000*` și vedem că programul se segfaultuiește. Folosim `gdb --args ./vuln_test crashes/id:000000...` pentru debug. Gdb arată că segfault-ul se întâmplă la scrierea în buffer-ul intern pentru linia citită, deci clar e un **buffer overflow**.
- Pas 5: Remediem codul: înlocuim funcția nesigură (de ex. `gets()` sau `strcpy` fără limită) cu una sigură (`fgets` cu limită, sau folosim `std::string` dacă era C++). Recompilăm binarul (de data asta normal sau tot cu AFL).
- Pas 6: Putem rula AFL din nou sau rula testul problematic – vedem că acum crash-ul nu mai are loc. Pentru a fi siguri, putem lăsa AFL să ruleze o rundă întreagă din nou: dacă nu mai găsește crash-uri după epuizarea mutațiilor, înseamnă că bug-ul a fost rezolvat.

Notă: În exemplul de mai sus, programul era clar vulnerabil. În cazuri reale, poate aveți un parser de fișier (ex. parser de imagini BMP). Puteți folosi AFL să-l testați: dați un BMP valid minim ca seed, și lăsați-l să fuzz-uiască. AFL ar putea descoperi un fișier BMP malformat care cauzează o citire dincolo de buffer (vulnerabilitate ce ar putea duce la execuție de cod dacă ar fi exploatată). AFL vă oferă acel fișier pe tavă pentru analiză. Multe vulnerabilități în software cunoscut au fost descoperite cu AFL în acest mod.

Exemplu real: SQLite (motorul de baze de date) a fost fuzz-uit cu AFL; au fost descoperite multiple crash-uri și probleme. AFL a generat automat inputuri SQL bizare care au dat peste cap parserul SQLite, evidențiind bug-uri. Dezvoltatorii au folosit aceste date pentru a repara codul și a întări securitatea. AFL este deci foarte util în **testarea fuzz** în special a codului care procesează date de intrare complexe (format de fișier, protocol etc.).

5. Interpretarea rezultatelor și remedierea vulnerabilităților

Acum că am rulat aceste unelte și am obținut rapoarte, vom discuta cum să interpretăm **rezultatele** lor și, mai important, cum să **remediem vulnerabilitățile** găsite. De asemenea, vom oferi bune practici pentru a preveni apariția unor astfel de probleme pe viitor.

5.1 Analiza rapoartelor și identificarea problemelor

- **Valgrind (Memcheck):** Rezultatul vine sub forma mesajelor în consola Valgrind. Cheia este să te uiți la tipul erorii și la call stack-ul furnizat:
    - *Invalid read/write*: indică acces ilegal la memorie. De regulă, Valgrind arată exact linia de cod din program unde se face accesul și, dacă e out-of-bounds, arată și unde s-a alocat acea memorie. Interpretarea: ai fie un indice greșit la un array (off-by-one), fie iterezi prea mult, fie folosești un pointer deja eliberat.
    - *Use of uninitialized value*: Valgrind detectează dacă folosești conținut nerulat (neinițializat). Stack-ul va arăta unde se utilizează valoarea și eventual unde a fost alocată. Acest lucru indică necesitatea de a inițializa variabile (ex. `memset` pe structuri, sau asigurarea că toate ramurile de cod atribuie o valoare înainte de folosire).
    - *Memory leaks* (leak summary): `definitely lost` înseamnă memorie pierdută sigur (nu mai există referințe la ea) – trebuie eliberată, e clar o scurgere. `possibly lost` de obicei înseamnă pointerul a fost modificat (ex. pointer spre mijlocul buffer-ului, deci adresa de start s-a pierdut). `still reachable` înseamnă memorie încă referită la final – nu e neapărat leak (poate fi memorie statică sau intenționat neeliberată la final), dar e bine s-o eliberezi și pe aceasta pentru curățenie. Ca remedieri: adaugă `free` lipsă, folosește smart pointers sau restructurează codul astfel încât deconstruirea să elibereze resursele.
    - *Invalid free / Mismatched free*: indică faptul că s-a apelat free de două ori pe același pointer sau pointerul nu e începutul unui bloc alocat. Aceasta e grav (double free poate duce la exploatări). Remediere: verifică logica de dealocare să nu eliberezi deja ceva eliberat (poate marchează pointerul NULL după free ca să știi că e liber).
    - *Cond. jump or move depends on uninit value*: adică un if sau calcul folosește o variabilă neinițializată, similar cu use of uninit dar subliniază că logica programului poate deraia din cauza ei.

- *Invalid opcode or segfault in Valgrind:* dacă Valgrind însuși raportează probleme majore, posibil ai hit un bug foarte sever (stack coruption) sau folosești o bibliotecă necompatibilă. Acestea sunt rare.

Prioritizează erorile: În general, abordează întâi *Invalid read/write* și *Invalid frees* – acestea cauzează instabilitate direct. Apoi scurgerile de memorie. După fiecare set de modificări, rerulează Valgrind până obții *ERROR SUMMARY: 0 errors*.

- **AFL (fuzzing):** AFL nu dă un „raport" textual clasic, ci rezultatele sunt:
  - **Status Screen:** Îți dai seama în timp real câte crash-uri și unde se află (in `out/crashes`). De obicei, după ce oprești AFL, te uiți în directorul de output.
  - **Fișierele de crash/hang:** Fiecare fișier input din `out/crashes` este un test-case minim (AFL tinde să minimizeze, deci nu e nevoie să rulezi tu afl-tmin decât dacă vrei extra). Numele fișierului conține codul de semnal: `sig:06` e abort, `sig:11` e segfault, etc. Poți deschide fișierul în hex/text să vezi conținutul. De multe ori e util să vezi ultimii bytes, uneori conține pattern-uri repetate (AFL are mutator de pattern repetitiv).
  - **Reproducerea crash-urilor:** Trebuie să rulezi manual programul cu acel fișier (sau pe input de la stdin dacă e cazul) într-un debugger. AFL îți dă input-ul, dar nu îți spune exact ce a cauzat – deci tu ca dezvoltator deschizi fișierul, rulezi în gdb: `run < id:00000X` și vezi unde se duce. Odată ce ai stack trace și linia de cod a crash-ului, repari bug-ul.
  - **Caz special – hang (timeout):** Dacă ai fișiere în `hangs`, înseamnă că programul intră în bucle infinite sau pur și simplu nu termină în timpul dat. Acelea sunt semnale de probleme de performanță sau bucle infinite. Trebuie analizate separat – de ex. un anumit input cauzează un algoritm să degradeze (O(n^2) greu). Soluția poate fi optimizarea algoritmului sau impunerea unui timeout intern/oprit procesarea inputului la un moment.
  - **Analiza acoperirii:** AFL++ oferă utilitarul `afl-plot` care poate genera un grafic (necesită gnuplot) cu progresul fuzzing (cum crește corpus-ul, cum scade rata de descoperire de noi paths). Asta e util pentru a decide când un fuzzing a „terminat" (dacă nu se mai descoperă nimic nou mult timp).
  - **No crashes but still issues:** Atenție, dacă AFL nu găsește crash, nu înseamnă că programul e 100% sigur, dar indică o anumită robustețe față de inputurile mutate. Ca o verificare în plus, puteți compila programul cu AddressSanitizer și rula AFL. Poate descoperi erori de memorie care altfel nu duc la crash imediat (ASan va

aborta programul la corupție). Aceste aborturi vor fi detectate de AFL ca crash (sig:06).

- După ce faceți fixuri, rulați AFL din nou eventual (sau testați manual crash-urile salvate pentru a vedea că acum programul nu mai moare pe ele). Puteți folosi `afl-cmin` ca să vă păstrați o suite de test minimală care acoperă toate codurile – adică inputurile unice. Asta poate fi integrată ca regresion testsuite.

Concluzie: Folosirea instrumentelor ca Valgrind, AFL ajută de a rezola o gamă largă de probleme de securitate și calitate. Fiecare are domeniul său:

- *Valgrind/DrMemory* pentru integritatea memoriei în aplicații native.
- *AFL* pentru robustețea la inputuri neașteptate (fuzzing).

În final, prin efectuarea lucrării de laborator trbuie obținute abilități de a

- Configura un mediu Linux pe Windows (WSL sau VM) pentru a rula unelte de testare.
- Instala și utiliza Valgrind pentru a găsi bug-uri de memorie și interpreta raportul (identificând memory leaks, invalid accesses și corectându-le).
- Utiliza AFL (sau AFL++) pentru a descoperi intrări care duc la crash-uri în programele, automatizând astfel testarea la nivel de intrare.

Surse (pentru consultare suplimentară): Valgrind Quick Start Guide, documentația AFL