Programowanie niskopoziomowe – Lab 5 – Exploit writing/PWN + Profilowanie

Exploit Writing / PWN

Exploit development (to po polsku najczęściej „tworzenie exploitów" lub „inżynieria podatności") jest dziedziną bezpieczeństwa informatycznego, która zajmuje się **przekształcaniem znalezionej podatności w działający, powtarzalny sposób jej wykorzystania**. Innymi słowy: odkrywamy błąd w oprogramowaniu, a następnie konstruujemy fragment kodu lub sekwencję danych – *exploit* – która przełamuje kontrolowane przez system ograniczenia (np. uzyskuje wyższe uprawnienia, wykonuje nieautorytatywny kod, czyta cudze dane).

Typowy cykl pracy exploit-dewelopera

Faza	**Krótki opis**	**Narzędzia / umiejętności**
Recon & triage	Zebranie informacji o programie, wersji, kompilatorze, ochronach.	file, checksec, IDA, Ghidra
Fuzzing / analiza bug-crash	Automatyczne lub ręczne wywoływanie funkcji z losowymi danymi; wychwycenie awarii.	AFL++, libFuzzer, honggfuzz
Debugging & root-cause	Odtworzenie warunków błędu, zrozumienie korzenia problemu (UAF, BOF, integer overflow).	GDB + pwndbg, WinDbg, lldb
Budowa prymitywów	Zamiana crasha w kontrolowany efekt: zapis pod adres, info leak, EIP/RIP control.	techniki ROP, format-string write, heap feng-shui
Bypass mitigation	Omijanie DEP, ASLR, canary, NX, PIE, relro	ROP, ret2libc, JOP, sigreturn
Polerowanie & stabilizacja	Utrwalenie exploitu, skrócenie łańcucha, dodanie reliablity checks.	scripting (Python/pwntools), CI fuzz-replay
Payload / post-exploitation	Shell, exfiltracja, privesc.	reverse shells, Meterpreter, custom stagers

Kategorie najpopularniejszych exploitów (poziom „low-level”)

1. **Stack-based buffer overflow** – nadpisanie adresu powrotu, ROP chain, shellcode.
2. **Heap overflow / Use-After-Free** – uzyskanie zapisu/odczytu arbitralnego, tzw. „arb-write”.
3. **Format string vulnerability** – wyciek adresów, zapis pod wskazany adres.
4. **Integer overflow / truncation** – przekroczenia zakresu, które łamią obliczenia rozmiarów bufora.
5. **Kernel / driver exploits** – eskalacja uprawnień z ring 3 → ring 0.
6. **Logic flaws** – np. błędne uwierzytelnienie, błędy wyścigów (TOCTOU).

Anatomia eksploitu

Element	**Opis**
Podatności (Vulnerabilities)	Błędy, luki lub niewłaściwe mechanizmy kontroli. Mogą wynikać z bugów programistycznych, złej konfiguracji, słabych zabezpieczeń lub „funkcji ubocznych”.
Techniki eksploatacji	Konkretne metody przekucia podatności w efekt. • **Buffer overflow** – nadpisanie pamięci poza buforem i wykonywanie własnego kodu. • **ROP (Return-Oriented Programming)** – łączenie istniejących instrukcji („gadżetów”) w pamięci, by ominąć zabezpieczenia i wykonać kod. • **Exploity sterty (heap)** – korupcja dynamicznej pamięci prowadząca do zapisu / wykonania dowolnego kodu.
Shellcode / payload	*Shellcode* – mini-program uruchamiany po udanym ataku, nierzadko otwierający powłokę. *Payload* – właściwa „złośliwa treść”, np. backdoor, ransomware, kradzież danych. Shellcode jest nośnikiem, payload – celem.

CVEs – standard nazewnictwa podatności

- **CVE (ID wzorca CVE-YYYY-NNNN)** – globalny katalog podatności prowadzony przez MITRE.
- Korzyści:
    - ujednolicona komunikacja (jedna nazwa na całym świecie),
    - centralne źródło wiedzy o łatkach, workaroundach,
    - możliwość priorytetyzacji dzięki **CVSS** (ocena krytyczności),
    - analiza historyczna i prognozowanie trendów.

Popularne narzędzia

Kategoria	**Przykłady**
Frameworki	**Metasploit** – otwarto-źródłowy, tysiące exploitów i payloadów; *Meterpreter* jako potężny shell.
Bazy exploitów	**Exploit-DB** – archiwum kodów + white-papers; użyteczne przy researchu i w CTF-ach.
Debuggery	**GDB** (Linux), **OllyDbg / x64dbg** (Windows) – podstawowe narzędzia analizy wykonywania, rejestrów, pamięci.
Fuzzery	AFL++, libFuzzer, honggfuzz – automatyczne generowanie nietypowych danych w celu wywołania crashy i znalezienia luk.

Metasploit

- **Najpopularniejszy framework do testów penetracyjnych** – projekt open-source (licencja BSD, właściciel Rapid7).
- Udostępnia **zbiór gotowych exploitów, payloadów, skanerów i pomocniczych narzędzi**; ułatwia automatyzację faz Red Team / Blue Team (walidacja podatności, proof-of-concept, post-exploitation).
- Składa się z - **Ruby-owego core** (msf-core), bazy modułów (modules/), powłoki **msfconsole**, API RPC/REST, GUI (*Armitage*, *Cobalt Strike* oparte na MSF), a w wersji komercyjnej – **Metasploit Pro** (web-GUI, raporty, automatyczne kampanie).

Typy modułów

Katalog	Cel	Przykład
exploit/	wykorzystanie podatności (RCE, LPE…)	windows/local/cldflt_cloud_files_lpe – CVE-2024-30085 (Rapid7)
auxiliary/	skanery, enumeracja, DoS, fuzzing	gather/glpi_inventory_plugin_unauth_sqli – CVE-2025-24799 (Rapid7)
post/	czynności po zyskaniu sesji	zbieranie hashy, eskalacja uprawnień
payload/	właściwy kod wykonywany na ofierze	linux/x64/meterpreter/reverse_tcp
encoder/	obfuskacja payloadu	x86/shikata_ga_nai
nop/	wypełniacze (NOP sled)	x86/single_byte
evasion/	unikanie AV/EDR (eksperymentalne)	windows/meterpreter_deflate

Podstawowy workflow

```
msfconsole              # start powłoki
msf6 > search cve:2025-24799
msf6 > use auxiliary/gather/glpi_inventory_plugin_unauth_sqli
msf6 auxiliary(...) > set RHOSTS 192.0.2.10
msf6 auxiliary(...) > run
```

1. **Wybór modułu** (use).
2. **Konfiguracja opcji** (show options, set).
3. **Sprawdzenie** check (jeśli moduł wspiera).
4. **Eksploitacja** run / exploit.
5. **Sesja** (sessions -i <ID>, meterpreter >) i post-exploitation.

Meterpreter

- Ładuje się w pamięci (brak pliku na dysku).
- Dynamicznie doładowuje rozszerzenia (stdapi, priv, extapi, ...); maj 2025 → łatka na *clipboard monitoring* CVE w extapi (Rapid7).
- Komendy: getuid, hashdump, migrate, screenshare, portfwd, railgun (WinAPI).
- Tryby łączności: TCP/HTTP/HTTPS bind / reverse, stager/stage-less, pivot przez route.

Pisanie własnego modułu

```
class MetasploitModule < Msf::Exploit::Remote
 Rank = NormalRanking
 include Msf::Exploit::Remote::Tcp
 def initialize(info = {})
  super(update_info(info,
   'Name'       => 'Demo BOF',
   'Author'     => 'you',
   'DisclosureDate' => '2025-05-12',
   'License'    => MSF_LICENSE,
   'Targets'    =>
    [ ['Linux x86', { 'Ret' => 0xdeadbeef, 'Offset' => 112 }] ],
   'DefaultTarget' => 0))
  register_options([ Opt::RHOST(), Opt::RPORT(1337) ])
 end
 def exploit
  connect
  payload = make_nops(target['Offset']) + [target['Ret']].pack('V')
  sock.put(payload)
  handler
  disconnect
 end
end
```

Profilowanie aplikacji

Valgrind powstał na początku wieku (pierwsze publiczne wydanie 2002 r.) jako **framework do dynamicznej instrumentacji** uruchamianych programów. Kod użytkownika nie wykonuje się bezpośrednio na procesorze; binarka ELF/ Mach-O/ ... jest ładowana do jądra Valgrinda (LibVEX), tłumaczona na **wewnętrzną reprezentację VEX IR**, a następnie instrukcje IR są symulowane krok po kroku. Na każdym kroku może interweniować wybrane narzędzie (*tool*) – odczytując stan pamięci, rejestrów, liczników, zegara, itd.

Dzięki tej architekturze:

- **nie wymaga rekompilacji** (w przeciwieństwie do AddressSanitizera) – wystarczy uruchomić gotowy plik wykonywalny,
- działa na szeregu architektur (x86, x86-64, ARM32/64, MIPS32/64, PPC, s390x, RISC-V 64, ...) oraz systemach Linux, FreeBSD, Solaris i macOS (amd64 + arm64) – lista w wydaniu 3.25.0 z 25 kwietnia 2025 r.(valgrind.org),
- umożliwia pisanie **własnych narzędzi** (Valgrind Plug-ins), jeśli wbudowane analyzery nie wystarczą.

Pakiet narzędzi (tool suite)

Narzędzie	**Cel analizy**	**Najważniejsze komunikaty / pliki wynikowe**
Memcheck	błędy pamięci, wycieki, użycie niezainicjowanej pamięci	Invalid read/write, definitely lost, Conditional jump ...
Cachegrind	symulacja L1/L2/L3, BTB; statystyki I-cache/D-cache, przewidywania skoków	cg.out.<pid>
Callgrind	profilowanie wywołań (na bazie wydarzeń Cachegrinda) – używane z KCachegrind/QCachegrind	callgrind.out.<pid>
Massif	szczytowe zużycie sterty („heap profiler")	massif.out.<pid> + massif-visualizer
DHAT	*Dynamic Heap Analysis Tool* – identyfikacja krótkich, często alokowanych obiektów	raport w json/tekst
Helgrind	wykrywanie wyścigów / synchronizacji w wielowątkowych programach POSIX	Possible data race during read of size ...
DRD	alternatywa do Helgrinda – mniej fałszywych alarmów, większy koszt czasowy	podobne ostrzeżenia

Instalacja i kompilacja programu

Większość dystrybucji Linuksa ma pakiet valgrind; w macOS → brew install valgrind
(na Apple silicon wymagana jest gałąź master). Do funkcji eksperymentalnych lub najnowszych architektur (np. RISC-V 64) trzeba zbudować wersję 3.25.0 ze źródeł (valgrind.org).

Kompilacja kodu

```
gcc -g -O0 -Wall -Wextra src.c -o prog       # debug info + brak optymalizacji
```

- **-g** – debug-symbole (bez nich Valgrind poda tylko adresy).
- **-O0** – łatwiejsze mapowanie adres→linia; Memcheck działa również dla -O2/-O3, lecz część linii może być „przemielona" przez optymalizacje.

Uruchamianie – najważniejsze przełączniki

```
valgrind \
 --tool=memcheck \
 --leak-check=full \
 --show-leak-kinds=all \
 --track-fds=yes \
 --track-origins=yes \
 --verbose \
 ./prog  arg1 arg2
```

- --leak-check=[no|summary|yes|full] – poziom dokładności analizy wycieków.
- --show-leak-kinds=[definite|possible|reachable|all] – które klasy bloków raportować.
- --track-fds=yes – raport otwartych plików/socketów przy wyjściu.
- --track-origins=yes – dla niezainicjowanej pamięci podaje **pierwotne źródło** (30–70 % wolniej).
- --trace-children=yes – śledzenie procesów potomnych po fork()/exec().
- --child-silent-after-fork=yes – ukryj raporty dzieci, dopóki nie wystąpi błąd (czytelniejsze logi).
- --suppressions=file.sup – plik z regułami ignorowania znanych „fałszywych pozytywów".

Kategorie wycieków (Leak Kinds)

Klasa	Definicja (Valgrind)	Typowy powód
definitely lost	brak wskaźnika do **początku** bloku; brak możliwości free()	zapomniany free(), zerowanie globala, utrata head listy
indirectly lost	blok osiągalny wyłącznie z bloku „definitely lost”	lista węzłów, której nagłówek został utracony
possibly lost	istnieje wskaźnik **do wnętrza** bloku, ale nie na offset 0	inkrementacja wskaźnika w pętli, iterator STLa wskazujący środek bufora
still reachable	wskaźnik znajduje się w żyjącej globalnej zmiennej lub na stosie przy wyjściu	singleton, pamięć cache, tablica globalna
suppressed	blok ignorowany wg reguł w .sup	znane wycieki w libc, OpenSSL, …

Profilowanie wydajności (Cachegrind & Callgrind)

- **Cachegrind** emuluje pamięć podręczną (domyślnie L1i 32 kB, L1d 32 kB, L2 256 kB) i BTB. Wyniki można analizować:
    - liczba brakowań Ir, Dr, Dw,
    - procent instrukcji vs cykli.
- **Callgrind** włącza licznik wywołań i krawędzi graficznych; plik callgrind.out.<pid> otwieramy w KCachegrind/QCachegrind → płomieniowy wykres, hotspoty.
- **Massif** mierzy „żywą stertę w czasie”, potrafi wskazać jednorazowe piki; wizualizacja: ms_print lub massif-visualizer.

Przykład praktyczny

Rozważmy następujący program:

```
1  #include <stdlib.h>
2
3  int main( int argc, char** argv) {
4    int i;
5    void* ptr;
6
7    for(i=0;i<20;i++) {
8      ptr = malloc(10 * sizeof(int));
9    }
10
11   free(ptr);
12   return 0;
13 }
```

Problem 1: (nie)zwalnianie pamięci II

Program pozostawia 19 niezwolnionych bloków pamięci w momencie powrotu z funkcji *main*.

Przykład 1 Kompilacja programu i uruchomienie Valgrinda

```
$ gcc -O0 -ggdb leak1.c -o leak1
$ valgrind ./leak1
```

Należy pamiętać aby skompilować program z symbolami debugowania. *Memcheck*jest Valgrinda, w związku z czym można pominąć opcję *--tool*

Problem 1: (nie)zwalnianie pamięci III

Przykład 2 Możliwe wyjście

```
==215== Memcheck, a memory error detector
==215== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et
al.
==215== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==215== Command: ./problem1
==215==
==215==
==215== HEAP SUMMARY:
==215==     in use at exit: 760 bytes in 19 blocks
==215==   total heap usage: 20 allocs, 1 frees, 800 bytes allocated
==215==
==215== LEAK SUMMARY:
==215==    definitely lost: 760 bytes in 19 blocks
==215==    indirectly lost: 0 bytes in 0 blocks
==215==      possibly lost: 0 bytes in 0 blocks
==215==    still reachable: 0 bytes in 0 blocks
==215==         suppressed: 0 bytes in 0 blocks
==215== Rerun with --leak-check=full to see details of leaked memory
==215==
==215== For counts of detected and suppressed errors, rerun with: -v
==215== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

W powyższym wyjściu można zauważyć dwie ważne informacje:

- *definitely lost: 760 bytes in 19 blocks*
- *with --leak-check=full to see details of leaked memory*

Pierwsza informacja mówi, że nie zostało zwolnionych 19 bloków z malloc lub pochodnych. W sumie 760 bajtów. Druga informacja to sugestia uruchomienia narzędzia z opcją wykonującą dodatkową analizę. Po uruchomieniu z dodatkową opcją:

```
$ valgrind --leak-check=full ./leak1
```

Otrzymujemy wynik zawierający informacje o miejscu alokacji niezwolnionych bloków.

```
==216==  760 bytes in 19 blocks are definitely lost in loss record 1
==216==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
==216==    by 0x400567: main (in/home/jan/git/unix/valgrind/leak1)
```

Więcej przykładów można znaleźć pod adresem:
https://pages.mini.pw.edu.pl/~karwowskij/unix/valgrind.pdf

perf to **standardowy pakiet narzędzi do analizy wydajności w Linuksie**, czerpiący dane prosto z jądra poprzez **perf events** (znane też jako *Performance Counters for Linux, PCL*) (Red Hat Documentation). W odróżnieniu od statycznych profilerów (np. gprof) czy narzędzi z instrumentacją (ASan, Valgrind), perf:

- wykorzystuje **sprzętowy PMU** (Performance Monitoring Unit) procesora,
- może rejestrować **tracepointy jądra**, liczniki programowe (page faulty, przełączenia kontekstu) oraz zdarzenia eBPF,
- działa z minimalnym narzutem (kilka % przy samplingu),
- udostępnia bogaty zestaw komend do pomiaru „na żywo" i post-mortem.

Warstwa	**Rola**
perf_event_open(2)	Syscall tworzący deskryptor zdarzenia; pozwala jądrze skonfigurować licznik, bufor mmap i filtry (Kernel.org).
Kernel perf events	Abstrakcja PMU + tracepointów; zarządza multiplexingiem liczników i buforami ring.
Program perf	Interfejs CLI (podkomendy stat, record, top, …).
Front-endy	perf report (ncurses TUI), perf script (dump do tekstu), graficzne nakładki (Hotspot, FlameGraph).

Uprawnienia – niektóre zdarzenia wymagają roota lub obniżenia /proc/sys/kernel/perf_event_paranoid.

Składnia ogólna

```
perf [global-opcje] <podkomenda> [opcje_podkomendy] [--] program [args]
```

Najważniejsze podkomendy (Kernel.org):

Komenda Perf	Cel
perf list	lista wszystkich dostępnych zdarzeń (PMU, trace, BPF).
perf stat	**liczniki sumaryczne** (cykle, instrukcje, LLC-missy...).
perf record	próbkowanie (sampling) i zapis do perf.data.
perf report	interaktywna analiza pliku perf.data.
perf top	podgląd „na żywo" (podobny do top, ale wg PMU).
perf script	konwersja perf.data do tekstu lub Python/Lua.
perf annotate	skorelowanie statystyk z asemblacją i krokiem źródeł.

perf stat

- Domyślnie zbiera **7 klasycznych metryk**: cykle, instrukcje, IPC, page-faulty itp.
- -e cycles,cache-misses – własny zestaw zdarzeń.
- -r <N> – powtórz pomiar (uśrednianie).
- --metric-only / --csv / --json – łatwa integracja z CI (np. benchy).
- Tryb system-wide: perf stat -a (uwaga na multiplexing).

Instalacja i źródła informacji

- Pakiet **linux-tools-$(uname -r)** (Debian/Ubuntu) lub perf w RHEL/Fedora (Red Hat Documentation).
- Dokumentacja: man perf, man perf-stat, man perf-record (Kernel.org, Kernel.org).
- Wiki: https://perf.wiki.kernel.org.
- Blogi ekspertów (np. Brendan Gregg) – obszerne zbiory przykładów (brendangregg.com).

Materiały dodatkowe

Exploit Developement and PWN

- https://docs.metasploit.com/
- https://www.exploit-db.com/
- https://pwn.college/
- https://github.com/jesusgavancho/TryHackMe_and_HackTheBox/blob/master/Intro%20To%20Pwntools.md
- https://pwn.college/program-security/reverse-engineering/
- https://pwn.college/program-security/memory-errors/
- https://pwn.college/program-security/shellcode-injection/
- https://pwn.college/system-security/race-conditions/
- https://pwn.college/software-exploitation/format-string-exploits/
- https://pwn.college/software-exploitation/file-struct-exploits/
- https://pwn.college/software-exploitation/memory-mastery/
- https://pwn.college/pwntools~9b09c9dc/pwntools/
- https://medium.com/@zeshanahmednobin/title-a-beginners-guide-to-pwntools-aaf56fc62e0a
- https://medium.com/@kuldeepkumawat195/mastering-pwntools-a-python-library-for-ctf-challenges-and-exploit-development-8547668c861c
- https://tryhackme.com/room/introtopwntools

Profilowanie:

- https://pages.mini.pw.edu.pl/~karwowskij/unix/valgrind.pdf
- https://www.brendangregg.com/perf.html
- https://web.stanford.edu/class/cs107/resources/valgrind
- https://www.youtube.com/watch?v=2tzdkC6IDbo&list=PLRWO2AL1QAV6bJAU2kgB4xfodGID43Y5d
- https://is.umk.pl/~grochu/wiki/doku.php?id=zajecia:npr:wyklad:profiler
- https://kompendium.plgrid.pl/analiza-wydajnosci/narzedzia/perf/
- https://www.redhat.com/en/blog/perf-vs-gprofng
- https://medium.com/@harshiljani2002/linux-perf-profiling-227d17101bd6

Zadania PWN/ Exploit Writing

Zadanie 1 – Analiza snippetów kodu

https://github.com/Sptimus/Low-level-programming/blob/main/Lab%205/Link.txt

Polecenie:

1. Przeanalizuj dostarczone snippet'y kodu zawierające przykłady:
    - kodowania
    - szyfrowania (np. XOR, AES),
    - obfuskacji danych.

Zadanie 2 – Jak złośliwe oprogramowanie unika wykrycia przez AV?

Zaimplementuj wybrany sposób unikania wykrycia (np. stager, xor, zaciemnianie kodu).

Utwórz prosty dropper lub loader, który wykorzystuje wybraną technikę. (można wykorzystać gotowy https://github.com/chvancooten/OSEP-Code-Snippets)

Przetestuj wykrywalność przygotowanego pliku na VirusTotal.

Opisz:

- jakie elementy kodu wpływają na wykrywalność,
- co mogłoby ją dodatkowo zmniejszyć,
- jakie są ograniczenia Twojej metody.

Zadanie 3 – PWN101

https://tryhackme.com/room/pwn101

Polecenie:

1. **Przejdź przez pokój PWN101 na TryHackMe.**
2. **Uzupełnij wszystkie sekcje oraz wykonaj zadania praktyczne z eksploitacji.**
3. **Zainstaluj i skonfiguruj bibliotekę pwntools.**
4. **Napisz krótki opis:**
    - **czym jest pwntools,**
    - **do czego służy,**
    - **przykład użycia w prostym ataku typu buffer overflow.**

Zadania dodatkowe z profilowania

Zadanie 1

Skompiluj następujący program a następnie przeanalizuj go z wykorzystaniem valgrind.

```
#include <stdlib.h>

void f(void)
{
  int* x = malloc(10 * sizeof(int));
  x[10] = 0;      // problem 1: heap block overrun
}               // problem 2: memory leak -- x not freed

int main(void)
{
  f();
  return 0;
}
```

Zadanie 2

Skompiluj i przeanalizuj następujący program z wykorzystaniem valgrind.

```
Oto przepisany kod programu z obrazka:
/* problem6.c */
#include <stdlib.h>

void* buff1;

int main(int argc, char** argv) {
  void* buff2;

  buff1 = malloc(1024);
  buff2 = malloc(1024);

  return 0;
}
```

Wykorzystaj przełączniki valgrind--leak-check=full oraz --show-leak-kinds=all i porównaj wyniki.

Zadanie 3

Zapoznać się z kursem: https://github.com/dendibakh/perf-ninja?tab=readme-ov-file

https://github.com/dendibakh/perf-ninja/blob/main/GetStarted.md