Przykładowe Tematy Projektów

1. **Analiza kodu binarnego (reverse engineering) aplikacji lub sterowników**
   – Wybór niewielkiego programu lub sterownika i przeprowadzenie pełnej analizy przy użyciu narzędzi takich jak IDA Pro, Ghidra czy radare2.
   – Identyfikacja funkcji, mapowanie przepływu sterowania, identyfikacja potencjalnych zabezpieczeń (obfuskacja, szyfrowanie). Przygotowanie szczegółowego raportu.

2. **Przeprowadzenie analizy złośliwego oprogramowania**.

   – Przygotowanie bezpiecznego środowiska (np. sandbox, maszyna wirtualna) do uruchomienia próbki złośliwego kodu.
   – Analiza statyczna (przegląd sekcji pliku, sygnatur, wskaźników na metody packowania, identyfikacja potencjalnych stringów).
   – Analiza dynamiczna w kontrolowanych warunkach (monitorowanie zachowania malware, rejestrowanie interakcji z systemem, badanie zmian w rejestrze/bazie danych).
   – Opracowanie raportu zawierającego opis działania złośliwego oprogramowania, zastosowane techniki zaciemniania oraz rekomendacje dotyczące wykrywania i neutralizacji.

3. **Reverse Engineering aplikacji mobilnej**

   – Analiza struktury aplikacji (np. .apk, .ipa), w tym pliku manifestu, zasobów oraz użytych bibliotek.
   – Dekompilacja kodu przy użyciu narzędzi (takich jak JADX, apktool, Ghidra) i wstępna interpretacja logiki aplikacji.
   – Identyfikacja istotnych elementów (mechanizmy autoryzacji, szyfrowania danych, kluczowe funkcje).
   – Przygotowanie raportu opisującego główne mechanizmy działania aplikacji oraz potencjalne podatności.

4. **Implementacja prostej obfuskacji w celu unikania wykrycia przez oprogramowanie antywirusowe**.

   – Wybór kodu (np. fragmenty w C/C++ lub innym języku wysokiego poziomu), który będzie poddany obfuskacji.
   – Zastosowanie podstawowych technik (m.in. zmiana nazw symboli, zaciemnianie przepływu sterowania, ukrywanie stringów).
   – Testowanie skuteczności obfuskacji przy użyciu różnych silników antywirusowych, analiza wyników wykrywania.
   – Opracowanie dokumentacji i wniosków dotyczących efektywności wybranych metod obfuskacji oraz możliwości ich ulepszenia.

5. **Optymalizacja kodu asemblerowego w kontekście specyficznych algorytmów**
   – Implementacja algorytmów (np. sortowania, operacji na macierzach) w NASM oraz ich porównanie z wersjami w C/C++ pod kątem wydajności.
   – Analiza wpływu optymalizacji na szybkość wykonania.

6. **Integracja kodu asemblerowego z kodem C/C++**
   – Projekt polegający na napisaniu modułu w NASM, który realizuje krytyczne operacje (np. algorytmy kryptograficzne) i łączenie go z aplikacją w C/C++.
   – Analiza interfejsu (ABI) i porównanie wydajności implementacji wysokopoziomowej i niskopoziomowej.

7. **Prosty kalkulator w C z wstawkami asemblerowymi**
   – Stwórz aplikację w C, która realizuje podstawowe operacje arytmetyczne, a kluczowe operacje (np. dodawanie, mnożenie) zaimplementuj przy pomocy inline assembly.

8. **Analiza wygenerowanego kodu asemblera**
   – Napisz kilka prostych funkcji w C (np. pętla for, instrukcja warunkowa, funkcja rekurencyjna), skompiluj je z różnymi poziomami optymalizacji i przeanalizuj wygenerowany kod asemblera (przy pomocy flagi -S i narzędzi takich jak objdump). Przygotuj raport i opisz istotne różnice.

9. **Program demonstrujący przekazywanie argumentów do funkcji**
   – Stwórz aplikację w C lub C++, która wywołuje kilka funkcji z różnymi typami argumentów, a następnie prześledź, jak te argumenty są przekazywane przez rejestry lub stos (analiza przy pomocy GDB/LLDB).

10. **Implementacja własnego alokatora pamięci**
    – Napisanie prostego alokatora pamięci w języku C, który zarządza stertą, implementując mechanizmy podziału, łączenia wolnych bloków i minimalizacji fragmentacji.
    – Porównanie wydajności i efektywności z biblioteką standardową.

11. **Analiza mechanizmu wywołań systemowych (syscalls) w systemie Linux**
    – Badanie, jak kod w C/C++ tłumaczony jest na wywołania systemowe (int 0x80, sysenter, syscall) w zależności od architektury (32-bit vs. 64-bit).
    – Porównanie wygenerowanego kodu asemblera przy użyciu różnych ABI. Przygotowanie raportu z analizy.

12. **Sumowanie elementów dużej tablicy**

Opis: Napisz program, który wczytuje lub generuje dużą tablicę liczb (np. miliony elementów) i oblicza sumę wszystkich elementów.

- **C++:** Wykorzystaj pętle for lub funkcje biblioteczne, a przy okazji sprawdź wersję zoptymalizowaną z flagą -O2 lub -O3.
- **NASM:** Zaimplementuj pętlę, która sumuje elementy tablicy, starając się wykorzystać rejestry.

13. **Obliczanie ciągu Fibonacciego**

Opis: Zaimplementuj iteracyjną (lub rekurencyjną) funkcję obliczającą n-ty element ciągu Fibonacciego.

- **C++:** Porównaj wersję rekurencyjną (bez optymalizacji) z iteracyjną lub wykorzystującą memoizację.
- **NASM:** Napisz wersję iteracyjną, przydzielając zmienne do rejestrów i eliminując niepotrzebne operacje pamięciowe.

14. **Sortowanie dużej tablicy**

Opis: Wybierz algorytm sortowania (np. quicksort lub sortowanie przez wstawianie) i zaimplementuj go dla dużej tablicy liczb.

- **C++:** Wersja wysokopoziomowa z wykorzystaniem standardowych konstrukcji językowych i optymalizacji kompilatora.
- **NASM:** Napisz ręcznie pętlę sortującą, dbając o optymalizację operacji porównania i zamiany elementów.

15. **Prosty algorytm szyfrowania (XOR encryption)**

Opis: Zaimplementuj prosty algorytm szyfrowania, który przetwarza blok danych przy użyciu operacji XOR.

- **C++:** Wersja, która przetwarza dane w pętli, odczytując i zapisując bajty, z możliwością porównania wyników i czasu wykonania.
- **NASM:** Napisz kod asemblerowy, który przetwarza dane na poziomie bajtów lub słów, starając się maksymalnie zoptymalizować pętlę.

Projekty dla ambitnych 😊

16. **Tworzenie narzędzia do automatycznej analizy binariów**
    – Opracowanie skryptu lub programu, który korzystając z objdump, radare2 lub Ghidry, generuje raporty dotyczące struktury pliku wykonywalnego, wykrywa sekcje, symbole oraz potencjalne podatności.

17. **Implementacja bootloadera lub prostego systemu operacyjnego**
    – Projekt polega na napisaniu bootloadera w NASM, który uruchomi minimalny kod systemu operacyjnego.
    – Zadania: obsługa BIOS/UEFI, ładowanie jądra, inicjalizacja podstawowych urządzeń.

18. **Własny interpreter asemblera lub emulator procesora**
    – Opracowanie narzędzia, które odczytuje i interpretuje instrukcje asemblera, symulując działanie procesora.
    – Można zaimplementować emulator dla uproszczonej architektury (np. subset x86) lub stworzyć interpreter dla języka asemblera.

19. **Własny debugger lub narzędzie monitorujące wykonywanie kodu**
    – Implementacja prostego debuggera w C/C++ lub NASM, który potrafi w czasie rzeczywistym śledzić rejestry, stos oraz wykrywać błędy (np. nieprawidłowe wywołania funkcji, naruszenia ochrony pamięci).