Zaawansowane metody debugowania w systemie Windows i Visual Studio

W programistycznej karierze przychodzi w końcu taki moment, w którym musimy zmierzyć się z wyjątkowo trudnym do zdiagnozowania błędem. Moją piętą achillesową okazał się bug, z którym walczyłem półtora tygodnia i poległem; dopiero moi koledzy, którzy zaczęli mozolnie komentować duże partie kodu, dotarli do źródła problemu. W niniejszym artykule przedstawię zbiór mniej lub bardziej zaawansowanych i niekonwencjonalnych technik debugowania.

SŁOWEM WSTĘPU

Walcząc przez długie lata z różnymi błędami obecnymi w rozwijanym przeze mnie oprogramowaniu, w moim metaforycznym podręcznym zasobniku z narzędziami do debugowania zebrałem cały szereg różnych technik i narzędzi, które pomagają przy rozpracowaniu nawet najbardziej złośliwych błędów. Znajdują się tam rozwiązania pomagające w bardzo różnych scenariuszach – dlatego też artykuł ten należy traktować bardziej jako kompendium, niż studium konkretnego przypadku lub pojedynczego, określonego zagadnienia. I o ile podejrzewam, że najwięcej zaczerpną z niego początkujący adepci sztuki programowania, być może i stary wyjadacz znajdzie tu coś dla siebie.

KŁOPOTY Z PAMIĘCIĄ

Błąd, o którym wspomniałem we wstępie, był naprawdę paskudny. Wiele lat temu oprogramowaliśmy z kolegą w C++ autorski, oparty na DirectX 11 silnik renderujący grafikę 3D, przy pomocy którego napisaliśmy potem zaawansowane narzędzie wizualizujące dane. I to właśnie przy zamykaniu tejże wizualizacji aplikacja zatrzymywała się na błędzie typu Access Violation (czyli naruszenia dostępu do pamięci), a co dziwniejsze – wewnątrz destruktora klasy `std::vector`.

Oczywiście natychmiast wykluczyliśmy możliwość błędnej implementacji tak fundamentalnej funkcjonalności, jaką jest `std::vector`, więc na kolejny ogień poszedł nasz wizualizator oraz silnik. Ale tak się złożyło, że był to akurat jeden z najsolidniej zaprojektowanych i zaimplementowanych mechanizmów w mojej karierze i tam również nie znaleźliśmy żadnego błędu.

Gdy po długiej, wyczerpującej i niestety bezowocnej walce w końcu się poddałem, koledzy zastosowali metodę, którą można określić sposobem ostatniej szansy – zaczęli komentować kolejne fragmenty aplikacji, sprawdzając za każdym razem, czy błąd wciąż występuje. I w ten sposób, po kilku dniach, znaleźli w końcu winowajcę.

Okazało się, że w zupełnie innym module, napisanym również w C++, a odpowiedzialnym za zbieranie danych z urządzenia, jedna z tablic indeksowana była beztrosko przy pomocy typu wyliczeniowego. Niestety podczas jej konstrukcji rozmiar podawany był w postaci stałej, co oczywiście skończyło się katastrofalnie, gdy do typu wyliczeniowego doszedł kolejny element.

Pikanterii sytuacji dodaje fakt, iż wspomniana tablica znajdowała się mniej więcej pośrodku klasy. W efekcie wadliwy kod spowodował, że nadpisane zostało pole znajdujące się bezpośrednio za wspomnianą tablicą – sytuację tę demonstruje kod z Listingu 1.

Listing 1. Nieprawidłowe indeksowanie tablicy

```
struct Test
{
    unsigned char tab[4];
    int value;
};

int main()
{
    Test test;
    test.value = 0x11223344;

    for (int i = 0; i <= 4; i++)
        test.tab[i] = 0xAA;

    printf("%x", test.value);

    getchar();
}
```

W wyniku błędnego indeksowania tablicy poza jej zakresem, zamiast oczekiwanego `11223344`, na ekranie zobaczymy `112233AA`.

Oczywiście sytuację tę można zdiagnozować, zaglądając do zrzutu pamięci w Visual Studio. W pierwszej kolejności musimy skorzystać z okna Immediate, które w momencie, gdy aplikacja jest zatrzymana, pozwala na wykonanie dowolnego kodu. Najpierw wpisujemy tam `&test`, aby otrzymać dokładny adres w pamięci zmiennej `test`. Następnie otwieramy okno „Debug | Windows | Memory" 1 i w polu adresu wpisujemy wcześniej uzyskaną wartość – w moim przypadku było to `0x0000002D3059FA18`. Po wykonaniu kilku kroków programu możemy zaobserwować, co dzieje się z pamięcią – na Rysunku 1 przedstawiony jest właśnie ten moment, w którym zamazywany jest pierwszy bajt pola `value`. Wartości w pamięci, które zostały zmienione przez program od poprzedniego jego zatrzymania, kolorowane są na czerwono.

Dodajmy jeszcze, że w przypadku rozwijanej przeze mnie wówczas aplikacji nadpisane pole, które znajdowało się zaraz za tablicą, reprezentowało rozmiar jakichś dynamicznie zaalokowanych danych. Nietrudno domyśleć się, jakie konsekwencje musiało mieć jego zamazanie.

Analizowanie zmian pamięci jest dobrym pomysłem, ale w przypadku demonstracyjnego programu z Listingu 1 wiedzieliśmy już, czego się spodziewać, i mogliśmy w łatwy sposób dostrzec moment nadpisania danych. Co więcej, wartości wpisywane do pamięci były łatwo rozpoznawalne (`11`, `22`, `33`, `44`, `AA`) – w rzeczywistości znalazłyby się tam znacznie bardziej „losowe" liczby.