Mateusz "j00ru" Jurczyk

Jak napisać własny debugger w systemie Windows – część 1

Jedną z rzadziej używanych funkcjonalności będącej częścią WinAPI jest tzw. Debug API, a więc zestaw wszystkich narzędzi niezbędnych do zbudowania własnego, w pełni sprawnego debuggera. Zadaniem niniejszego artykułu jest zapoznanie czytelnika z możliwościami oferowanymi przez ów interfejs, oraz zachęcenie do spróbowania swoich sił na polu tworzenia narzędzi, które monitorują i kontrolują przebieg działania innych aplikacji w systemie. Zachęcam do dalszej lektury!

WSTĘP

Kiedy większość początkujących programistów słyszy o nauce Windows API (w skrócie WinAPI), przed oczami staje im najczęściej interfejs odpowiadający za tworzenie, obsługę oraz niszczenie okien trybu graficznego oraz znajdujących się na nich kontrolek. O ile skojarzenie to jest częściowo poprawne – odpowiednio zaprojektowane i zaimplementowane GUI jest w końcu bardzo istotnym elementem większości aplikacji działających pod systemem Windows – gigant z Redmond nie ograniczył się wyłącznie do udostępnienia developerom możliwości wyświetlania okien. Windows API obejmuje w istocie zbiór wszystkich udokumentowanych stałych, struktur i publicznie dostępnych funkcji, które sprawiają, że program może wchodzić w interakcję z systemem operacyjnym w dowolny sposób: operować na systemie plików, rejestrze, komunikować się z innymi programami, otwierać i korzystać z połączeń sieciowych itd.

Jedną z takich funkcjonalności jest właśnie interfejs umożliwiający monitorowanie i ingerowanie w stan innych procesów w systemie, którym zajmujemy się w tym artykule.

WŁASNY DEBUGGER?

Zanim podejmiemy się nauki technicznych aspektów związanych z pisaniem własnego debuggera, warto zadać sobie zasadnicze pytanie – w jakim celu (oprócz oczywistego elementu edukacyjnego) miałby on powstać i jakiemu celowi służyć? Faktycznie, uważny czytelnik zauważy, że programista w dzisiejszych czasach może wybierać z szerokiej gamy narzędzi pomagających śledzić wykonywanie programów w systemie Windows i odszukiwać znajdujące się w nich błędy; do najpopularniejszych zaliczyć można debugger dołączony do środowiska Microsoft Visual Studio (w dowolnej wersji), WinDbg (również autorstwa Microsoftu), OllyDbg, gdb należący do pakietu MinGW gcc czy też debugger będący integralną częścią znanego disassemblera IDA. Po co więc komu kolejny projekt tego typu?

Nawet jeśli pominiemy fakt, że wiele z podanych wyżej rozwiązań jest dotkniętych pewnymi niedoskonałościami (niewygodny lub nieistniejący interfejs graficzny, brak istotnych opcji itp.), to okazuje się, że wciąż istnieje kilka dobrych powodów, dla których warto rozważyć posiadanie własnoręcznie napisanego debuggera w swoim informatycznym arsenale. O ile do tej pory termin „debugger" był używany wyłącznie w kontekście programu, który umożliwia człowiekowi badanie i ingerowanie w stan aktywnych procesów, Debug API daje w rzeczywistości znacznie większe możliwości. Na przykład, przy jego użyciu wykonalne staje się stworzenie dynamicznych *unpackerów* (programów rozpakowujących), odwracających działanie cienkiej warstwy kompresującej i/lub szyfrującej dane pliku EXE, nierzadko utrudniającej tym samym proces jego analizy i używanej (między innymi) przez autorów złośliwego oprogramowania. Nasz własny debugger może okazać się również przydatny do automatycznego generowania raportów na temat powodu, z którego awaryjnie zamknęła się inna, testowana aplikacja, np. jako część większej infrastruktury umożliwiającej testowanie stabilności i bezpieczeństwa programów przez dostarczanie im losowych, niepoprawnych danych na wejściu – technika znana szerzej pod angielską nazwą *fuzzing*. Innym sposobem na wykorzystanie potencjału drzemiącego w Debug API może być zastosowanie go jako specyficznej formy zabezpieczenia naszego własnego oprogramowania przed obejściem zaimplementowanej logiki licencyjnej, lub przed możliwością oszukiwania w przypadku gier komputerowych.

Jak widać, możliwości skutecznego użycia opisywanego w artykule interfejsu jest wiele, a ich dokładna ilość ograniczona jest tylko przez wyobraźnię każdego z nas. Jedno jest pewne – wiedza na temat sposobu działania debuggerów w Windows może przydać się każdemu programiście, który tworzy lub zamierza tworzyć z myślą o tym systemie, w szczególności zaś, jeśli towarzyszy mu fascynacja niskopoziomowymi aspektami informatyki. Zabierzmy się więc za stworzenie swojego pierwszego debuggera z prawdziwego zdarzenia! Wszystkie listingi przytoczone w kolejnych sekcjach zostały napisane w języku C++.

WPROWADZENIE DO DEBUG API

Zanim napiszemy pierwsze linie kodu naszego programu, musimy wyjaśnić kilka terminów oraz podstawowych zasad obowiązujących w systemie Windows w związku z obsługą procesów oraz wzajemnymi relacjami pomiędzy debuggerem a programem debugowanym.

- Każdy proces w systemie Windows dzieli się na wątki (ang. *threads*), które z punktu widzenia debuggera są najmniejszą jednostką wykonania kodu. Każdy wątek posiada swój własny kontekst (zestaw rejestrów i flag procesora), lecz wszystkie wątki działające w kontekście tego samego procesu współdzielą przestrzeń adresową. Z definicji w skład każdego procesu wchodzi co najmniej jeden wątek – kiedy ostatni wątek ginie, niszczony jest cały proces.
- Każdy proces może być debugowany co najwyżej przez jeden, inny proces. Każda próba „podpięcia" się do procesu, który jest już debugowany, zakończy się niepowodzeniem. Z drugiej strony, jeden proces może debugować wiele różnych procesów naraz.
- Kod logiki debuggera musi wykonywać się w kontekście tego samego wątku debuggera, który podpiął się do danego procesu.

Zasady interakcji pomiędzy procesem debugowanym a debuggerem są bardzo proste. Dla debugowanego fakt podpiętego debuggera jest z zasady transparentny – wykonuje się on (a przynajmniej powinien) w taki sam sposób, w jaki wykonywałby się w normalnych warunkach. Wyjątek stanowią tutaj niektóre biblioteki systemowe, które w momencie wykrycia nadzorującego je procesu włączają pewne dodatkowe funkcjonalności umożliwiające łatwiejszą detekcję pewnych typów błędów. Za przykład może posłużyć tutaj alokator pamięci dynamicznej znajdujący się w bibliotece *ntdll.dll*, który w obecności debuggera wykonuje dodatkową weryfikację integralności wewnętrznych struktur sterty, szybko eksponując błędy takie jak np. przepełnienia bufora. Sama aplikacja również może sprawdzić, czy znajduje się

pod nadzorem innego procesu przy użyciu funkcji `IsDebuggerPresent` lub za pomocą wielu nieudokumentowanych sposobów [1][2][3], jednak jest to sytuacja rzadko spotykana i charakterystyczna prawie wyłącznie dla malware oraz programów, które w ten sposób próbują uchronić się przed piractwem; w większości przypadków debugowany proces w żaden sposób „nie zwraca uwagi" na to, że jest śledzony.

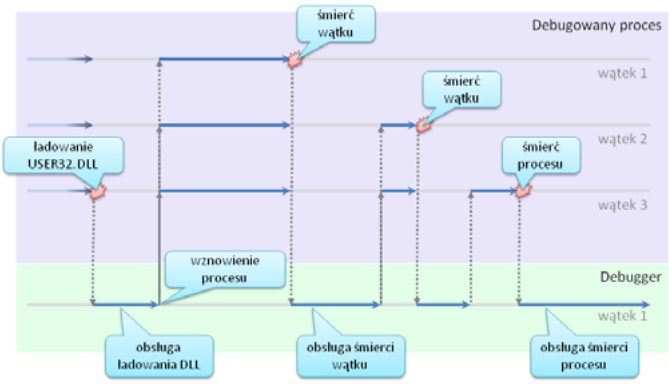
Działanie debugowanej aplikacji przerywane jest przez system operacyjny w przypadku wystąpienia jednego z następujących zdarzeń:

- proces rozpoczyna wykonywanie (właśnie został uruchomiony),
- w kontekście procesu został utworzony nowy wątek,
- podczas wykonywania jednego z wątków wygenerowany został wyjątek,
- proces jest zamykany (ostatni wątek w procesie zginął),
- dowolny wątek w procesie ginie,
- do przestrzeni adresowej procesu ładowany jest nowy moduł DLL,
- program przekazał inf. debuggerowi dzięki funkcji `OutputDebugString`,
- z przestrzeni adresowej procesu usuwany jest moduł DLL,
- proces ginie w nieoczekiwany sposób.

W momencie wykrycia dowolnego z powyższych warunków, wszystkie wątki debugowanego procesu zostają tymczasowo wstrzymane, a informacja o zdarzeniu przekazywana jest do debuggera, który może w tym momencie:

- zbadać dowolny element stanu debugowanego procesu i na podstawie zebranych informacji uaktualnić wewnętrzne struktury powiązane z tym procesem,
- w wybrany sposób zaingerować w stan programu, np. modyfikując pamięć procesu, kontekst dowolnego wątku, likwidując dowolny wątek lub zabijając cały proces itp.,
- wznowić wykonywanie procesu.

Uproszczony schemat interakcji pomiędzy debuggerem a debugowanym procesem został przedstawiony na Rysunku 1.



Rysunek 1. Uproszczony schemat interakcji pomiędzy debugowanym procesem a debuggerem na przykładzie końcowego etapu działania aplikacji

Na całokształt debuggera składają się trzy najważniejsze komponenty: kod odpowiadający za podpięcie się pod dany proces (w niektórych przypadkach – równocześnie z jego utworzeniem), główna pętla debuggera odpowiedzialna za oczekiwanie na informacje o zdarzeniach, obsługiwanie ich i wznawianie działania programu, a także część odpowiedzialną za odpięcie się od procesu wtedy, kiedy jest to konieczne. Kolejne sekcje opisują poprawny sposób implementacji każdego z tych elementów, ze szczególnym naciskiem na pętlę debuggera, będącej najważniejszą jego częścią, gdyż to tam znajduje się cała logika naszego programu.

PODSTAWY TWORZENIA DEBUGGERA

W tej sekcji krok po kroku stworzymy swój pierwszy debugger, szczegółowo opisując każdy z odrębnych elementów logiki naszej aplikacji. Zanim jednak przyjdzie nam monitorować i ewentualnie zmieniać stan innego procesu, musimy go wpierw utworzyć, lub dać systemowi operacyjnemu sygnał, że chcielibyśmy podpiąć się jako debugger do działającej już aplikacji.

Pierwsze kroki – podpinanie się do procesu

Niezależnie od tego, jaką skomplikowaną mechanikę zaimplementujemy w głównej pętli debuggera, pierwszym krokiem musi być prozaiczne zarejestrowanie jednego z wątków naszego programu jako debugującego inny proces. W celu utworzenia nowego, debugowanego procesu, konieczne jest przekazanie flagi `DEBUG_PROCESS` lub `DEBUG_ONLY_THIS_PROCESS` do szóstego parametru funkcji `CreateProcess` o nazwie `dwCreationFlags`, jak przedstawiono na Listingu 1.

Listing 1. Przykład funkcji odpowiedzialnej za tworzenie debugowanego procesu dziecka o zdefiniowanej ścieżce

```
BOOL SpawnAndAttachProcess(LPTSTR lpApplicationName) {
  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  ZeroMemory(&si, sizeof(si));
  ZeroMemory(&pi, sizeof(pi));
  si.cb = sizeof(si);

  if (!CreateProcess(lpApplicationName, NULL,
                     NULL, NULL, FALSE,
                     DEBUG_ONLY_THIS_PROCESS,
                     NULL, NULL, &si, &pi)) {
    return FALSE;
  }

  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);
  return TRUE;
}
```

Jedyną różnicą w semantyce obu wspomnianych flag jest sposób traktowania procesów potomnych pierwotnego programu, który debugujemy. Jeśli nasz proces-dziecko uruchomi w przyszłości kolejne programy, w razie użycia flagi `DEBUG_PROCESS` zostaną one automatycznie podpięte pod debugger procesu rodzica; w przeciwnym razie będą one wykonywane niezależnie. Innymi słowy – jeśli debugowany przez nas program nie posiada nigdy potomków w drzewie procesów (jak na przykład w przypadku aplikacji *calc.exe* lub innych prostych narzędzi), nie ma znaczenia, która z flag zostanie użyta. Jeśli tacy potomkowie mogą się pojawić, musimy zastanowić się, czy konkretne zastosowanie dla tworzonego debuggera wymaga informacji pochodzących z całego poddrzewa procesów, czy wyłącznie z jednej, konkretnej aplikacji.

Pomyślne wywołanie funkcji `CreateProcess` skutkuje wypełnieniem czterech pól struktury `PROCESS_INFORMATION`: `hProcess`, `hThread`, `dwProcessId`, `dwThreadId`, czyli kolejno uchwytów do procesu oraz jego pierwszego (głównego) wątku, a także ich globalne identyfikatory w systemie. Ze względu na fakt, że równoważne uchwyty zostaną nam przekazane podczas obsługi zdarzeń „nowy proces" i „nowy wątek", polecamy w tym momencie zamknąć oba uchwyty, aby uniknąć ich wycieku (zapomnienia) w dalszej części programu. Na tym etapie do niczego nam się one nie przydadzą.

Jeśli zakładamy, że program, który zamierzamy debugować, działa już w systemie (lub implementujemy podejście hybrydowe, podłączając się do istniejącego procesu lub tworząc nowy w przypadku jego braku), przydatna okazuje się funkcja `DebugActiveProcess`, która, jako jedyny parametr przyjmując identyfikator interesującego nas procesu, podłącza wywołujący ją wątek do owego procesu, umożliwiając jego normalne debugowanie – dla pętli debuggera zupełnie nieistotne jest, w jaki sposób nastąpiło podpięcie się do procesu. Funkcja `DebugActiveProcess` nie otwiera żadnych dodatkowych uchwytów, a jedynie tworzy w systemie „połączenie" między wątkiem a innym procesem.

W celu odnalezienia identyfikatora odpowiadającego konkretnej aplikacji, można posłużyć się jedną z wielu dostępnych w systemie Windows funkcji