Inline hook

Vložil/a RubberDuck [1], 28 Listopad, 2012 - 13:52

- Programming [2]

Článek popisuje, co je to obecně hook, co je to inline hook, jakým způsobem může programátor implementovat inline hooky a způsoby, jak předcházet špatné implementaci. Celý článek je doplněn o ukázkový kód inline hooku funkce MessageBoxA.

Co je to hook
Hook (česky háček) je technika zachycení volání funkce a její přesměrování na kód podstrčený jinou aplikací/uživatelem. Jednoduše řečeno: Pokud jsme schopni zjistit, že program volá požadovanou funkci z libovolného umístění v rámci jednoho nebo více procesů a jsme schopni i přesměrovat tok vykonávání programu na náš kód tak, aby zůstal plně konzistentní (aby nedošlo k havárii aplikace), jsme schopni pozměnit chování celého programu takovým způsobem, aby dělal přesně to, co chceme my. Příklad z reálného světa: Běžec běží orientační běh (vykonávání procesu). Za normálních okolností bude z kontrolního bodu 3 (nějaká funkce číslo 3) hledat další kontrolní bod relativně dlouho a může se rozloučit s dobrým umístěním (normální běh programu). Jenže u kontrolního bodu 3 na něj čeká kamarád, který mu nejen prozradí, kudy se nejrychleji dostat ke kontrolnímu bodu 4 (nějaká funkce 4, která musí být splněna, jinak program dál nepoběží), ale navíc mu řekne, že může rovnou pokračovat ke kontrolnímu bodu 5 (nějaká funkce 4), protože on již zařídil, že čip závodníka bude u kontrolního bodu 4 odpípnut za 4 minuty (změna toku/chování procesu/aplikace). Nyní se vraťme k procesům. Jediné, co tedy musí útočník podniknout, aby zhookoval/zaháčkoval funkci, je změnit kód programu na konkrétním místě tak, aby ukazoval na kód, jenž programu dodá sám. Existuje velké množství technik hookování a jejich variací. Již dříve jsem popisoval IAT (Import Address Table) hook [3] a EAT (Export Address Table) hook [4]. Kromě těchto dvou se nejčastěji setkáváme s inline hookem (dále v tomto článku) nebo s například exotickým SEH hookem (možná v některém z dalších článků).

Co je to inline hook
Inline hook je bezesporu nejuniverzálnější (je možné ho použít nejen v user modu a v kernel modu - rovněž ho lze využít i v jiných operačních systémech) a nejzákeřnější (pokud je napsán chytře, je velmi obtížně vystopovatelný - ale při troše snahy se vystopovat dá vždy) typ hooku. Existuje vícero implementací. V článku budou popsány dvě nejčastější a programově implementováno to nejjednodušší řešení. Celý koncept inline hooku je postaven na základní myšlence, že každá funkce má svůj kód a tento kód někde začíná a někde končí. Pokud na začátek kódu funkce umístíme skok na naši funkci, která nahradí existující kód funkce, můžeme ovlivnit, co bude funkce vracet, ale i to, jak se bude chovat. Pokud by to bylo všechno tak jednoduché, bylo by to prima. Jenže realita nám hází pod nohy klacky, Proto není možné vždy stejnou realizaci hooku.
Velká část API funkcí z DLL knihoven začíná tzv. prologem. Cílem prologu je vytvořit tzv. rámec funkce, což je vlastně označení adresy zásobníku/stacku, kde začíná paměť určená pro funkci. Prolog má následující kód (číslo za instrukcí udává tzv. opkód instrukce):

```
push ebp          55h
mov ebp, esp    8BECh
```

Výše zmíněný kód uloží obsah registru EBP na zásobník. Registr EBP (někdy se mu říká bázový) většinou slouží pro uchovávání bázové adresy funkce. Následně do registru EBP uložíme okamžitou hodnotu registru ESP. Registr ESP ukazuje na vrchol zásobníku. V tento okamžik tedy registry ESP i EBP ukazují na vrchol zásobníku - rámec funkce. Pokud kdekoliv dále v programu budeme chtít získat adresu začátku (báze/rámce) funkce, stačí si přečíst obsah registru EBP, pokud tento nebyl před tím ještě někde změněn. Podle opkódu výše zmíněného kódu víme, jakou má velikost. Je to 3 bajty (1

bajt za první řádek + 2 bajty za druhý) Pro zajímavost: Opakem prologu je epilog. Ten provádí zrcadlově obrácený proces:

```
mov esp, ebp
pop ebp
```

Nejdříve se do registru ESP uloží hodnota z registru EBP. Tím se vrchol zásobníku posune na bázovou/rámcovou adresu funkce a všechny hodnoty, které byly během vykonávání funkce umístěny na zásobník, jsou zahozeny (v reálu zahozeny nejsou, ale operační systém již neručí za jejich aktuálnost).

Aby byla situace ještě zamotanější, používal se prolog v tomto provedení pouze do Windows XP SP2. Pak ho Microsoft změnil následovně:

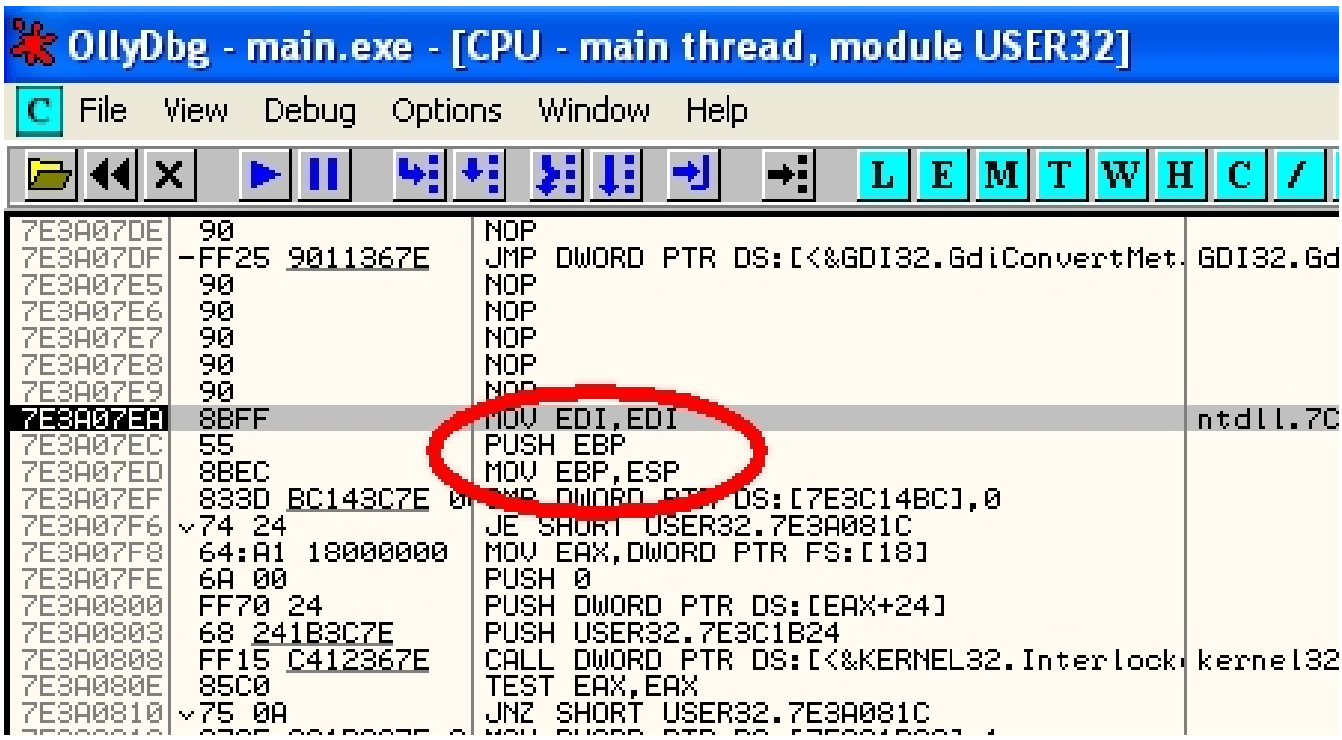
```
mov edi, edi     8BFFh
push ebp           55h
mov ebp, esp     8BECh
```

Měla tahle změna nějaký hlubší význam (kromě toho, že si tak Microsoft značně zjednodušil implementaci tzv. hotpatchů, jenž využívají právě inline hooking)? Vždyť tato instrukce nic nedělá a pouze nafukuje velikost prologu ze tří bajtů na pět. Přiřadit hodnotu z registru EDI do registru EDI znamená, že se hodnota v registru EDI nezmění a celý kód by mohl být tedy nahrazen dvojicí instrukcí NOP:

```
nop               90h
nop               90h
push ebp          55h
mov ebp, esp    8BECh
```

Takhle by kód vypadal ekvivalentně, ale nebylo by poznat, kde funkce skutečně začíná. Dalším důvodem je fakt, že téměř před každou API funkcí je pět instrukcí NOP, takže by byl v kódu o to větší zmatek.

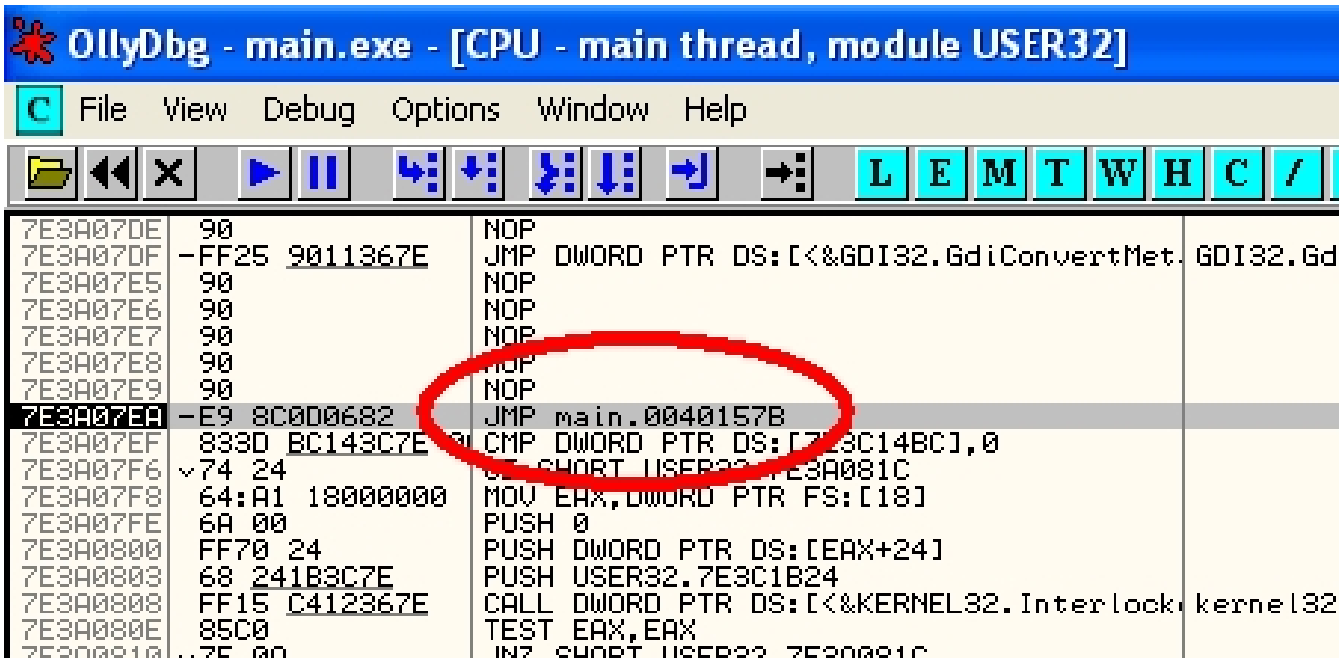
Možná se ptáte, k čemu se tady tak podrobně zabývám prologem. Ten je pro nás totiž důležitý. Víme, jak prolog vypadá, známe jeho velikost, a když si teď ověříme, jak velký je opkód pro instrukci long jmp addr, zjistíme, že je totožná s velikostí prologu (na Winodws počínaje XP SP2). Na systémech před Windows XP SP2, kde je velikost prologu pouze 3 bajty musíme provést jen short jmp a to na adresu prvního z pěti NOPů před funkcí. Zde můžeme následně uložit rovněž opkód pro long jmp addr (je možné implementovat ještě jedno řešení, které mě napadlo, ale tím se v tomto článku zabývat nebudu). Obecně je technika s krátkým odskokem mnohem lepší. Důvod? Přepisování pěti bajtů je na úrovni assembleru realizováno dvěma operacemi, protože v běžném x86 assembleru je možné v jeden okamžik zapsat maximálně 4 bajty. Pokud by se vykonávání programu dostalo do této části v okamžiku, kdy by nebyly ještě přepsány všechny potřebné bajty, může dojít k pádu aplikace. Jestliže ale nejdříve přepíšeme 5 NOPů nad funkcí a následně v jediném okamžiku přepíšeme část prologu kódem short jmp -5, jenž má velikost 2 bajty, tomuto problému se úplně vyhneme. Tento článek pro zjednodušení ukáže kratší variantu bez přepisování NOPů, čtenář si následně může v rámci domácího úkulu kód přepsat na verzi s krátkým skokem :)



[5]
Prolog funkce před přepsáním

Nyní můžeme skočit na libovolné místo v rámci kódu procesu. Dále potřebujeme vytvořit zástupnou funkci pro náš hook. Tato funkce sama o sobě může volat původní funkce a v závislosti na vrácené hodnotě reagovat. Pokud například někdo zkouší zjistit adresu naší DLL knihovny pomocí API funkce LoadLibrary, náš kód získá od originální funkce reálnou adresu, ale vrátí uživateli NULL, což znamená, že danou DLL knihovnu se z nějakého důvodu nepovedlo nahrát do procesu.

Tohle všechno zabalíme do jediné aplikace. Máme dvě možnosti: Buď vytvořit plnohodnotný process injector nebo celý kód vytvořit ve formě DLL knihovny a tu následně nainjektovat do požadovaného procesu pomocí DLL injectoru.



[6]
Prolog přepsaný skokem na daný offset

Teoretická realizace inline hooku

Teoreticky si teď popíšeme dvě řešení. První, náročnější, se většinou realizuje pomocí assembleru nebo inline assembleru ve vyšších jazycích (C/C++). Naše alternativa k zahookované funkci předpokládá znalost originální funkce a má schopnost znát velikost a tvar přepisovaného kódu (jak bylo zmíněno výše, ne vždy funkce začíná prologem a pokud bychom přepsali nesprávný počet bajtů kódu, výsledkem může být fatální změna původního kódu vedoucího v lepším případě k okamžitému pádu, v horším pak ke skryté chybě projevující se jen občas, při splnění specifických podmínek) ať už díky programátorovi nebo díky implementaci dissassembleru (program schopný přeložit opkódy zpět na assembler). Přepisovaný kód si hookovací funkce uloží na bezpečné místo. Když dojde k zavolání hookované funkce, nejprve se zavolá původní, přepsaný kód. Poté je proveden skok na začátek původní API funkce s adresním posuvem o počet přepsaných bajtů. Tím je zajištěno, že se původní kód vykoná v plném rozsahu, jako by funkce vůbec upravena nebyla. Následně je zpracován (a případně upraven) výsledek originální funkce a vrácen programu.

Aby byl kód srozumitelnější o pro neassembleristy, využijeme jednodušší řešení. Celý koncept zůstává prakticky stejný. Jen při přepisování kódu funkce se nebudeme vůbec zabývat jeho délkou ani případnou změnou výsledného kódu. Jednoduše si ho uložíme a přepíšeme. Když potom během vykonávání programu dojde k zavolání naší hooknuté funkce, jako první věc provedeme odhookování, tedy obrácený postup o proti hookování. Tím dojde k obnovení původní struktury kódu funkce. Následně zavoláme originální funkci a zpracujeme výsledek. Nyní provedeme opětovné zahookování funkce a výsledek vrátíme.

Až do tohoto okamžiku jsem zamlčoval jednu důležitou věc. Rozhodl jsem se věnovat se jí mimo hlavní popis jednoduše proto, že mnoho lidí na tuto věc zapomíná a následně se hloupě ptá, proč jim jejich kód nefunguje. Kód funkce je v tzv. sekci kódu. To je oblast paměti obsahující reálný kód programu. Sekce kódu má téměř vždy přístupová práva RW (readable writable - čtení vykonání/spuštění). Pokud se pokusíme zapsat kamkoliv vrámcí sekce kódu byť jen jediný bajt, výsledem bude stav ACCESS_VIOLATION a program skončí s chybou. Řešením je nastavení práva W (writable - zapisování) na všechny bajty kódu funkce, jež budeme přepisovat. Po přepsání této části paměti opět příznak W odebereme.

Praktická realizace inline hooku

Jako ukázkový příklad zahookujeme notoricky známou funkci Windows API funkci MessageBoxA v rámci našeho procesu. První věc, kterou vytvoříme bude funkce HookFunction. Funkce bude vracet hodnotu true nebo false v závislosti na tom, zda se hook zdařil nebe ne. Funkce bude brát tři argumenty:
- jméno DLL knihovny, ve které je funkce, jenž budeme hookovat, umístěna
- jméno hookované funkce
- adresa naší funkce nahrazující originální funkci

Prvním krokem bude získání handlu DLL knihovny. K tomu slouží API funkce GetModuleHandle za předpokladu, že je funkce již v procesu nahrána (musí tam být, jinak nemáme co hookovat ;)). Alternativně lze využít i API funkci LoadLibrary. Ale tady pozor! Pokud není knihovna v procesu, funkce LoadLibrary bude míst snahu tuto knihovnu nahrát do procesu. Další možností je vlastní implementace funkce GetModuleHandle procházející strukturu PEB daného procesu a vypisující informace o zavedených modulech. Pokud máme handle knihovny, můžeme se pokusit získat adresu funkce v ní. Zde využijeme API funkce GetProcAddress. Osobně bych ale volil radši vlastní implementaci funkce GetProcAddress [7], protože tato funkce může být zahookována někým jiným a nemusí vracet relevantní výsledky. Máme tedy adresu funkce. Volání funkce v rámci programu je většinou realizováno formou instrukce call na offset v rámci sekce kódu. Na tomto offsetu se nachází tzv. trampolína - nejedná se o nic jiného, než instrukci jmp na skutečnou adresu dané funkce. Důvod? Pokud by nebyla použita trampolína, během každého spouštění aplikace by loader musel procházet celou sekci kódu a doplňovat potřebné adresy, takže místo jedné adresy konkrétní funkce by jich mohlo být v kódu třeba dvacet, což není zrovna nejlepší. Offset spočítáme velmi jednoduše. Jedná se o rozdíl mezi adresou naší funkce a adresou originální funkce. Od této hodnoty ještě odečteme velikost našeho kódu, kterým budeme přepisovat původní kód originální funkce. Standardně by mělo stačit pět bajtů. Výše jsem zmiňoval problém se zápisem do sekce kódu. Použijeme tedy funkci VirtualProtect. Tato funkce změní přístupová oprávnění k bloku paměti daného adresou a množstvím bajtů. Doporučuji uložit si původní přístupová práva a po zapsání kódu nastavit původní práva danému kusu paměti. Ke zkopírování dat poslouží například funkce memcpy. Nejprve

si zálohujeme původní kód na bezpečné místo, následně vložíme opkód pro long jmp a na konec přidáme námi vypočítaný offset. Kód by mohl vypadat následovně:

```
bool HookFunction(char *szDllName, char *szFunctionName, LPVOID lpAddressFakeApi){
  HMODULE hDll = NULL;
  PBYTE pOriginalAddress = NULL;
  DWORD dwOldProtect = 0;
  DWORD dwJmp = 0;

  hDll = GetModuleHandleA(szDllName);

  if(hDll != NULL){
    pOriginalAddress = (PBYTE)GetProcAddress(hDll, szFunctionName);

    if(pOriginalAddress != NULL){
      dwJmp = (((PBYTE)lpAddressFakeApi - pOriginalAddress) - 5);

      VirtualProtect(pOriginalAddress, 10, PAGE_READWRITE, &dwOldProtect);
      memcpy(lpSavedBytes, pOriginalAddress, 5);
      memcpy(pOriginalAddress, "\xE9", 1);
      memcpy(pOriginalAddress + 1, &dwJmp, 4);
      VirtualProtect(pOriginalAddress, 10, dwOldProtect, &dwOldProtect);

      return true;
    }
  }

  return false;
}
```

Nyní máme funkci pro zahookování libovolné funkce. Teď si vytvoříme přesný opak, funkci pro odhookování. Funkce pro odhookování vypadá téměř totožně jako funkce pro zahookování jen s tím rozdílem, že přijímá pouze dva argumenty a že neukládá přepisovaný kód a není tedy třeba počítat offset. Argumenty jsou:
- jméno DLL knihovny, ve které je funkce, jenž budeme hookovat, umístěna
- jméno hookované funkce
Jedna z možných implementací je:

```
bool UnhookFunction(char *szDllName, char *szFunctionName){
  HMODULE hDll = NULL;
  PBYTE pOriginalAddress = NULL;
  DWORD dwOldProtect = 0;

  hDll = GetModuleHandleA(szDllName);
  if(hDll != NULL){
    pOriginalAddress = (PBYTE)GetProcAddress(hDll, szFunctionName);
    if(pOriginalAddress != NULL){
      VirtualProtect(pOriginalAddress, 10, PAGE_READWRITE, &dwOldProtect);
      memcpy(pOriginalAddress, lpSavedBytes, 5);
      VirtualProtect(pOriginalAddress, 10, dwOldProtect, &dwOldProtect);

      return true;
    }
  }
  return false;
}
```

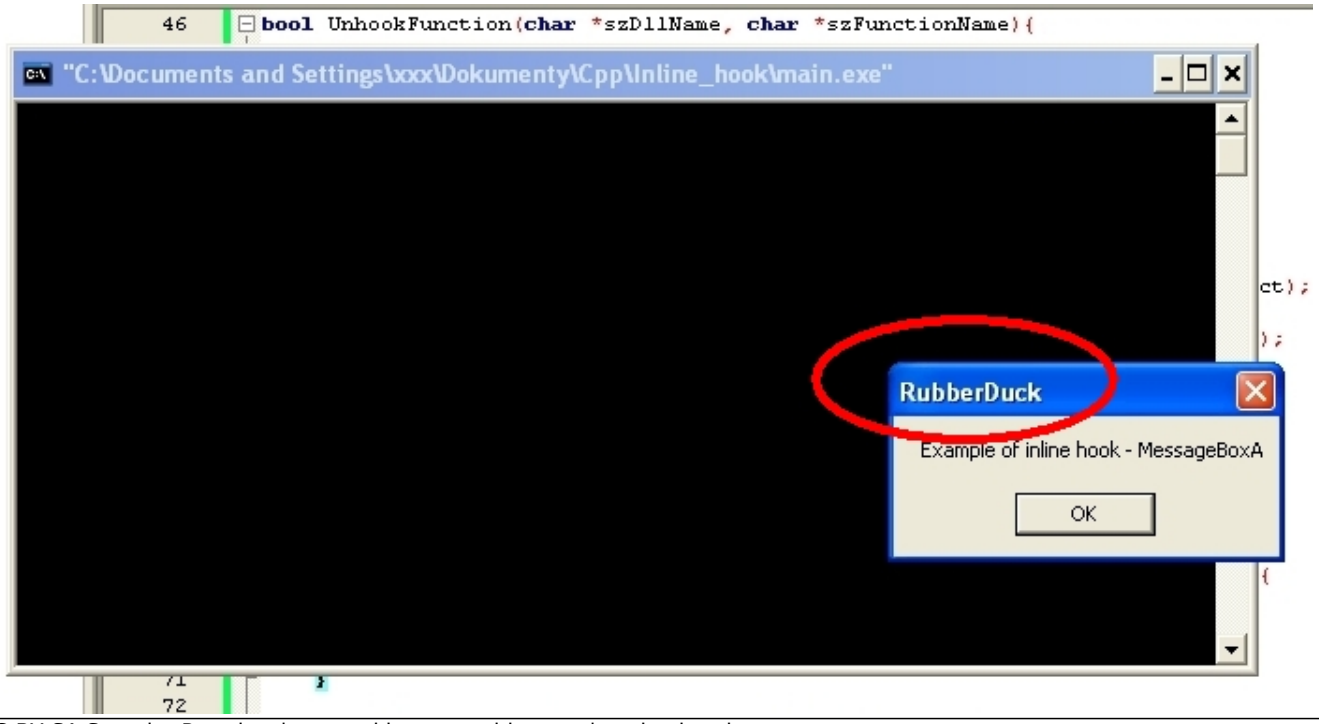
Máme funci pro zahookování, máme funkci pro odhookování. Nyní nás čeká poslední věc, naše náhradní funkce za zahookovanou funkci. Rozhodli jsme se zahookovat funkci MessageBoxA. Jako první věc se podíváme do dokumentace, jak vypadá prototyp funkce MessageBox:

```
int WINAPI MessageBox(
  _In_opt_  HWND hWnd,
  _In_opt_  LPCTSTR lpText,
  _In_opt_  LPCTSTR lpCaption,
  _In_      UINT uType
);
```

Naše funkce bude mít stejný prototyp, jen jiné jméno, řekněme MessageBoxX. V prvním kroku odhookujeme původní funkci MessageBoxA. Pokud se nám to povedlo, Zavoláme původní funkci MessageBoxA. Jako argumenty ji můžeme předat reálné hodnoty, jež jsme získali nebo můžeme použít libovolné vlastní. Řekněme, že budeme měnit titulek okna na text "Hooked!". Předáme tedy první, druhý, a čtvrtý argument a místo druhého vložíme požadovaný text. Vhodné je zachytit návratovou adresu funkce a tu nakonec i vrátit programu. Následně musíme funkci opětovně zahookovat. Kód může být následující:

```
int WINAPI MessageBoxX(HWND hWnd, LPSTR lpText, LPSTR lpCaption, UINT uType){
  DWORD dwRet = 0;
  if(!UnhookFunction("user32.dll", "MessageBoxA")){
    return 0;
  }

  dwRet = MessageBoxA(hWnd, lpText, "Hooked!", uType);

  if(!HookFunction("user32.dll", "MessageBoxA", (void *)(&MessageBoxX))){
    return 0;
  }

  return dwRet;
}
```

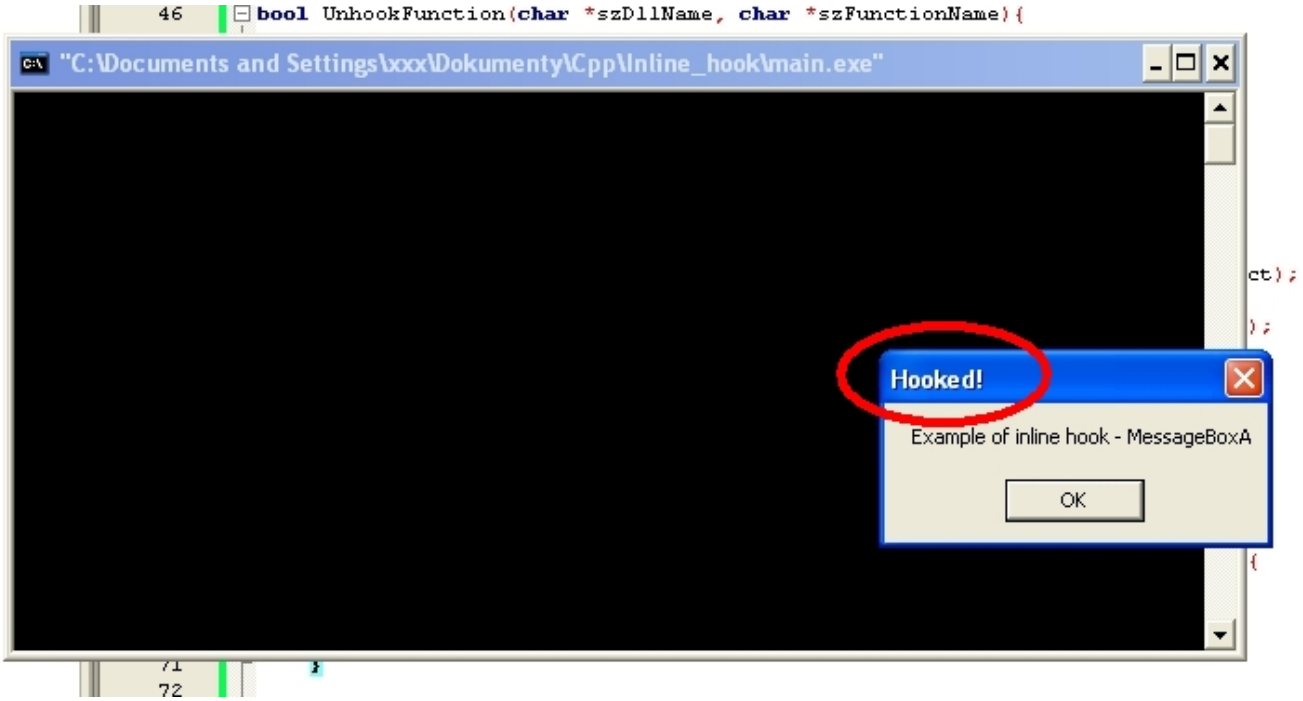
Tím všechna naše práce končí a my můžeme testovat výsledný kód :)



[8]
Originální MessageBox zobrazuje skutečná data

```
#include <windows.h>

bool HookFunction(char *DllName, char *FunctionName, LPVOID AddressFakeApi);
bool UnhookFunction(char *DllName, char *FunctionName);
int WINAPI MessageBoxX(HWND hWnd, LPSTR lpText, LPSTR lpCaption, UINT uType);

char lpSavedBytes[5];
LPVOID lpMessageBoxX = (LPVOID)&MessageBoxX;

int main(){
  MessageBoxA(NULL, "Example of inline hook - MessageBoxA", "RubberDuck", MB_OK);
  if(!HookFunction("user32.dll", "MessageBoxA", lpMessageBoxX) == true){
    return 0;
  }

  MessageBoxA(NULL, "Example of inline hook - MessageBoxA", "RubberDuck", MB_OK);

  return 0;
}

bool HookFunction(char *szDllName, char *szFunctionName, LPVOID AddressFakeApi){
  HMODULE hDll = NULL;
  PBYTE pOriginalAddress = NULL;
  DWORD dwOldProtect;
  DWORD dwJmp;

  hDll = GetModuleHandleA(szDllName);
  if(hDll != NULL){
    pOriginalAddress = (PBYTE)GetProcAddress(hDll, szFunctionName);
    if(pOriginalAddress != NULL){
      dwJmp = (((PBYTE)AddressFakeApi - pOriginalAddress) - 5);

      VirtualProtect(pOriginalAddress, 10, PAGE_READWRITE, &dwOldProtect);
      memcpy(lpSavedBytes, pOriginalAddress, 5);
      memcpy(pOriginalAddress, "\xE9", 1);
      memcpy(pOriginalAddress + 1, &dwJmp, 4);
      VirtualProtect(pOriginalAddress, 10, dwOldProtect, &dwOldProtect);

      return true;
    }
  }

  return false;
}

bool UnhookFunction(char *szDllName, char *szFunctionName){
  HMODULE hDll = NULL;
  PBYTE pOriginalAddress = NULL;
  DWORD dwOldProtect = 0;

  hDll = GetModuleHandleA(szDllName);
  if(hDll != NULL){
    pOriginalAddress = (PBYTE)GetProcAddress(hDll, szFunctionName);
    if(pOriginalAddress != NULL){
```

```
      VirtualProtect(pOriginalAddress, 10, PAGE_READWRITE, &dwOldProtect);
      memcpy(pOriginalAddress, lpSavedBytes, 5);
      VirtualProtect(pOriginalAddress, 10, dwOldProtect, &dwOldProtect);

      return true;
    }
  }

  return false;
}

int WINAPI MessageBoxX(HWND hWnd, LPSTR lpText, LPSTR lpCaption, UINT uType){
  DWORD dwRet = 0;
  if(!UnhookFunction("user32.dll", "MessageBoxA")){
    return 0;
  }

  dwRet = MessageBoxA(hWnd, lpText, "Hooked!", uType);

  if(!HookFunction("user32.dll", "MessageBoxA", lpMessageBoxX)){
    return 0;
  }

  return dwRet;
}
```



[9]
Zahookovaný MessageBox s přepsaným titulkem okna

Závěr
Technika inline hooku popsaná v článku je nejjednodušší formou. V praxi se lze setkat s mnohem sofistikovanějšími inline hooky využívající volná místa v paměťovém prostoru DLL knihovny (tzv. caves), čímž se jeví, že je kód součástí této DLL knihovny. Opravdu kvalitní inline hooky nepřepisují první bajty, ale jsou schopny umístit vlastní kód (někdy i několik desítek bajtů) za začátek takovým způsobem, aby neohrozili výstupní výsledek. Takové hooky se pak vyhledávají podstatně náročněji a programy, jež je využívají, mohou nepozorovaně v systému sídlit po mnoho týdnů, měsíců i let.

Originál: BFLOW [10]

URL článku: https://security-portal.cz/clanky/inline-hook

Odkazy:
[1] https://security-portal.cz/users/rubberduck
[2] https://security-portal.cz/category/tagy/programming
[3] http://bflow.security-portal.cz/import-address-table-hooking/
[4] http://bflow.security-portal.cz/export-address-table-hooking/
[5] http://security-portal.cz/sites/default/files/ihook_original.jpg
[6] http://security-portal.cz/sites/default/files/ihook_hooked.jpg
[7] http://bflow.security-portal.cz/vlastni-verze-windows-api-funkce-getprocaddress/
[8] http://security-portal.cz/sites/default/files/ihook_before.jpg
[9] http://security-portal.cz/sites/default/files/ihook_after.jpg
[10] http://bflow.security-portal.cz/inline-hook/trackback/