PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Nástroj pro reverse engineering

2011

Vladimír Matlach

Anotace

Software, který vznikl v rámci této práce, je určen k analýze malware, virů a reverznímu inženýrství. Umožňuje monitorování volání interních i externích funkcí zkoumaného programu, upravovat parametry a návratové hodnoty funkcí. Monitorování je založeno především na přepisování vlastního kódu monitorovaného programu tak, aby přeposílal informace do monitorovací aplikace. Výsledkem je software schopný poskytnout co nejlepší náhradu za klasický debugger v případě, že jej nelze použít. Software byl testován na řadě dnešních virů s velmi uspokojivými výsledky. Pokračováním v této práci by bylo možné vytvořit nástroj pro penetraci a testování stability software.

Děkuji Mgr. Petru Krajčovi Ph.D. za vstřícné vedení této práce, poskytnuté konzultace a za pomoc při korekci výsledného textu.

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

> They must find it difficult... Those who have taken authority as the truth, rather than truth as the authority.
>
> – Gerald Massey

Tato práce má za cíl vytvořit nový nástroj určený pro reverzní inženýrství a rozšířit tak spektrum nástrojů použitelných a zefektivňujících například analýzy virů – které denně zaplavují Internet a ohrožují jeho uživatele. Požadavky na takový software byly vytvářeny postupně z praktických zkušeností autora této práce a jeho několikaletých zkušeností v tomto oboru.

Aby bylo možné tento software hodnotit a vysvětlit důležitost jeho funkcí, věnuje se tato práce přímo i tématem reverzního inženýrství – od jeho úplných základů, až po ukázky, kdy běžné nástroje reverzního inženýrství selhávají a je nutné použít nový přístup, o který se tato práce pokouší, a demonstruje jeho výsledky.

Většina textu je koncipována takovým způsobem, aby byl snadno pochopitelný a srozumitelný i pro běžného programátora (tedy nejde o text výhradně určený pouze systémovým programátorům), veškerá problematika je nastíněna a vysvětlována relativně k její důležitosti, přičemž důraz je kladen spíše na správnou představu a principy jednotlivých technik, než na jejich konkrétní implementaci, která bude ovšem velmi detailní u samotné konečné implementace výsledného programu. Velkou výhodou pro čtenáře je ovšem stále alespoň povrchní znalost jazyka Assembler a některého z vyšších jazyků, např. jazyka C. Text se dále pokouší čtenáře vést tak, aby si sám uvědomoval různá úskalí a problémy, které mohou v reverzním inženýrství nastat – zde je nutné podotknout, že programování a reverzní inženýrství jsou do určité míry i věcí fantazie a čistě vědecký přístup nemusí být vždy tím nejideálnějším.

Text je koncipován následovně. Kapitola 1 se zabývá úvodem k této práci. Kapitola 2 se zabývá Úvodem do reverzního inženýrství, jeho definic, cílů, technik a využívaného software. Kapitola 3 navazuje na předchozí kapitolu a na konkrétních příkladech poukazuje na nedostatky a absolutní selhání používaného a veřejně dostupného software. Kapitola 4 shrnuje závěry o zjištěných nedostatcích a definuje požadavky na nový software. Kapitola 5 je pak věnována zkoumání nového konceptu fungování reverzního software a jeho následnou implementací.

Kontakt na autora:
email: v.matlach@seznam.cz

2. Úvod do technik reverzního inženýrství

V této první kapitole se seznámíme s tím, co přesně reverzní inženýrství je, co je jeho cílem, jaké používá techniky a jak se proti těmto technikám dá bránit.

2.1. Reverzní inženýrství, definice a pohledy

Reverzní inženýrství z anglického *reverse engineering* (RE) – zpětná analýza, zkoumání – by se dalo snadno definovat jako vědomá činnost, při které se snažíme pochopit fungování určitého již existujícího a fungujícího subjektu (ať už jde o novou technologii plazmových televizí, nového motoru, nebo počítačového software). Ve zkratce se tak jedná o snahu pochopit, jak byla daná věc vytvořena, jak funguje a nejlépe i to, jak byla vynalezena. Produktem úspěšného procesu reverzního inženýrství je pak schopnost vysvětlit principy fungování daného subjektu a následně i schopnost subjekt svépomocí replikovat a upravovat [1]. Tento text se bude konkrétně zabývat reverzním inženýrstvím počítačového software a především pak počítačových virů[1], které se většinou proti reverznímu inženýrství brání a znemožňují tak jeho běžné postupy. Reverzní inženýrství software má však i další využití, jakým je například obnovení starého odloženého software, doprogramovávání dalších funkcionalit do již hotového software atd.

Počítačové viry jsou z mnoha důvodů vhodným subjektem ke zkoumání. Jednak je záhodno vědět, co přesně daný virus dělá, aby bylo možné zamezit nebo co nejvíce minimalizovat škody při, nebo po infikaci (např. pokud virus krade bankovní certifikáty, tak okamžitě zjednat blokaci účtu atp.). Za druhé je velmi důležité zjistit, jak se virus šíří a jaké techniky používá[2], což je důležité z hlediska dalšího zabezpečení – aktualizování schopností a rozšíření ochran různého bezpečnostního software (HIPS, firewall, antivirus) nebo přímo opravení bezpečnostních chyb v systému.

Ovšem, jak již bylo nastíněno výše, cíle reverzního inženýrství nejsou vždy a pouze *ušlechtilé*. Reverzní inženýrství jako takové je dlouho známým jevem, který stojí za mnoha „nedobrovolnými (a často také nevědomými) předáními“ obchodních tajemství a patentů do rukou veřejnosti, popřípadě jen do rukou konkurence (v tomto případě jde o velmi výnosný byznys).
Dále se zaměřme pouze na software a ostatní technologie ponechme stranou. Většina počítačových uživatelů zná zajisté pojmy *crack, keygen nebo serial key*[3] [6], které jsou úspěšnými produkty reverzních inženýrů-pirátů, známých více jako

[1] Za počítačové viry budu dále v textu považovat veškerý software primárně určený jako nástroj pro provádění počítačové kriminality bez vědomí uživatele.

[2] Např. techniky skrývání sebe sama, infikace systémových souborů atp., o různých technikách dále v textu.

[3] Základní pojmy počítačového pirátství – crack: upravený soubor software tak, aby neobsahoval žádnou ochranu; keygen: generátor správných sériových čísel (serial key) nezbytných pro registraci software.

„crackeři“. Tyto produkty nepřináší pouze výhodu neplatit za daný software, ale na druhé straně přináší i nepříjemné důsledky ve formě finančních ztrát vydavatele – ti se následně snaží svůj software chránit pomocí různých, třeba i komerčních řešení, což přitahuje formou výzvy různé soutěživé crackery a nastává známý problém „kroku napřed“. Ovšem autoři virů si jsou také velmi dobře vědomi, že jejich software bude zkoumán a rozhodně si také nepřejí, aby byly jejich techniky a činnost kompromitovány zveřejněním. Dnešní viry tak používají mnoho velice sofistikovaných a velmi propracovaných technik, které brání jejich efektivní analýze a v některých případech zabrání analýze úplně (např. po dlouhou dobu neprolomitelná ochrana spustitelných souborů Themida).

2.2. Začínáme s RE

Než začneme s vysvětlováním postupů, je nutné zmínit, že celý tento text má pouze studijní charakter a jeho autor nenese žádnou zodpovědnost za zneužití jakýchkoliv uvedených informací nebo softwaru, k tomuto textu vytvořeném dle zadání.

Ačkoliv jsou následující postupy relativně univerzální na mnoha platformách, budeme se konkrétně věnovat pouze systému Microsoft Windows, a to z mnoha závažných důvodů – především pak z jeho značné oblíbenosti a rozšířeném použití ve státní správě, mezi uživateli, v armádě, ve školách, v korporacích a dalších zařízeních po celém světě. Přitom právě mnohé z těchto subjektů se stávají cílem profesionálních (občas i amatérských) snah o infiltraci, a to jak státních, tak soukromých špionážních agentur.

Začneme s upřesněním představ, které bychom měli mít. Program v jeho spustitelné podobě, ať už v podobě souborů s příponou exe, dll, ocx aj., je datovým souborem majícím určitou strukturu (dále konkrétně PE formát – Portable Executable) a samotná data udávající běh programu – instrukce procesoru, instrukce pro interpret a další data ve formě *resource*, ve kterých jsou uloženy obrázky, GUI dialogy a podobně. Je důležité mít na paměti, že kód = data a data = kód – právě tento důležitý fakt nám dává možnost nazpět „konvertovat“ (zpětně přeložit) již hotový program do jednoduše lidsky srozumitelné podoby (mnemonika atp.). Zpětnému překladu do mnemonické podoby assembleru říkáme *disassembling*, tento termín budeme v budoucnu používat právě pro proces překladu do assembleru, oproti jen *zpětnému překladu*, který budeme používat pro překlad instrukcí interpretovaných jazyků.

Nyní se podívejme na příklad ilustrující překlad zdrojového kódu do strojových instrukcí procesoru a následný překlad zpět:

```
Jazyk C:
if (a == 123) return;

Strojový kód:       Mnemonická podoba (disassemling):
83F8  7B       ->   CMP EAX, 0x7B (123)
75    01       ->   JNZ @skip
C3             ->   RET
                    @skip:

Překlad do vyššího jazyku lidským mozkem / programem:
if (EAX == 123) return;
```

Jak je z výše uvedeného příkladu vidět, dovolením program disassemblovat, nebo jej zpětně překládat, jednoduše umožňujeme zobrazovat celý zdrojový kód[4]. Aby byl tento proces co nejjednodušší a nejefektivnější, vznikla celá řada různých nástrojů – některé z nich přímo cíleně pro tyto účely, některé z nich byly pouze vhodně využity. Podívejme se tedy na tyto nástroje blíže.

2.2.1. Disassembler

Disassemblery, jak už může být patrné z názvu, jsou programy určené pro překlad strojových instrukcí do jejich mnemonické podoby. Jednotlivým instrukcím (jako je mov, jmp, call atd.) jsou přiřazeny číselné kódy výrobcem procesoru – disassembling tedy jako takový není ničím jiným, než pouhým procházením tabulky všech instrukcí zveřejněných výrobcem procesoru a hledání jejich jména. Např. z výše uvedeného příkladu 83F8 7B značí: 83 = CMP, F8 = kombinace EAX a hodnoty, 7B = hodnota 123. Časem si většina programátorů na tyto kódy zvykne a dokáží pak psát přímo strojovým kódem.

Disassemblery jako takové jsou určené ke *statické* analýze – zkoumání kódu programu, který neběží – je uložený pouze v paměti. Tato statická analýza by se dala snadno přirovnat k pitvě neznámého druhu živočicha (programu), kdy má patolog možnost vidět všechny jeho orgány (funkce, rutiny) a blíže zkoumat, co přesně dělají. Takové zkoumání může nabývat několika podob – od zcela povrchního typu odhadování „od oka“, až po zcela exaktní zkoumání každé instrukce, což bývá většinou velice náročné[5].

[4] A pokud byl program vytvořen jako Debug build, obsahuje i plná jména funkcí a proměnných.

[5] Už jen studium zdrojového textu programátora, který nepíše přímočaře a používá zbytečně dlouhé a nesmyslné konstrukce, je složité. Při RE je docela častým jevem zkoumání nesmyslných konstrukcí v kódu, protože se může například jednat o zcela novou a geniální myšlenku, kterou je potřeba odhalit.

Disassemblery nejsou žádnou výjimkou od obyčejných kancelářských aplikací a mnoho z nich má kvalitní GUI zpříjemňující práci. Jmenujme si hlavní dva zástupce disassemblerů:

W32Dasm (URSoftware Co.)
Tento disassembler zmiňuji snad už jen ze slušnosti a nostalgie, protože do příchodu IDA (viz dále) byl jedním z nejpoužívanějších disassemblerů vůbec a přinesl integraci mnoha užitečných nástrojů do jediné platformy. Ke dnešnímu dni je však po mnoha stránkách zastaralý a veřejně dostupný jako shareware za cenu 75 USD.

IDA – The Interactive Disassembler (Hex-Rays)
Dnešním dnem je IDA rozhodně jedním z nejprofesionálnějších a nejpůsobivějších nástrojů pro reverzní inženýrství vůbec. Dokáže analyzovat a překládat kód spustitelných souborů Windows, Linuxu, Maců, DOSu, lze ji dokonce použít i na další různé platformy od PDA, Windows Mobile, Symbian až po konzole typu PlayStation, Nintendo a desítky dalších.
Tím, čím je IDA ovšem opravdu neocenitelná, je její zpracování a dotažení uživatelského komfortu až k maximu. Tou největší lahůdkou je však i přídavný plugin Hex-Rays Decompiler, který se pokouší přímo o dekompilaci kódu – tedy o znovu vytvoření původního zdrojového kódu v pseudojazyce velice blízkém jazyku C. (Ovšem je nutné podotknout, že tento překlad ne vždy bývá správný a občas není možné kód přeložit.) IDA je navíc ke stažení zdarma na stránkách výrobce[6], ovšem s určitými omezeními platforem a některých schopností. Pro představu, jak se s IDA pracuje a jak vypadá běžná analýza vzorku viru, doporučuji shlédnout prezentační video (až tutorial) na adrese http://www.ccso.com/files/hexraysdemo.swf.

2.2.2. Debugger

Debugger zajisté zná každý programátor. Je to nástroj, s jehož pomocí je možné krokovat běh programu po „jednotlivých řádcích“ kódu nebo každé jednotlivé instrukci s možnostmi sledovat hodnoty proměnných, návratových hodnot funkcí a *call-stacku* (výpisu vnořených volání) atd. V kontextu zkoumání cizích programů je debugger neocenitelnou pomůckou v mnoha směrech – debuggerem zkoumáme běžící (živý) program – „nepitváme“ jej jako v disassembleru typu W32Dasm (ačkoliv W32Dasm i IDA obsahují vlastní debuggery, jsou však mnohem těžkopádnější, než k těmto účelům přímo určený a specializovaný software). Debugger nám zkrátka poskytuje komplementární informace, které pomocí *statické* analýzy v disassembleru získáme jen s obtížemi a nepřiměřenou námahou.

Důležité je také nastínit, jak debugger funguje. Jelikož je debugging (lovení

[6] http://www.hex-rays.com/idapro/

chyb) v programech zcela legitimní činností, obsahuje systém Windows řadu API funkcí[7] určených právě k těmto účelům – tzv. debug API. Debuggery skrze tyto funkce jednoduše ovládají běh debuggovaného procesu. Důležité je, jakým způsobem může debugger k cizím procesům přistupovat a co se s takovým procesem pak děje:

1. debugger spustí požadovaný proces, nebo se k němu „připne“ (attach),
2. systém změní v debuggovanému procesů různé příznaky (flagy) tak, aby umožnil bezproblémové debuggování a zároveň změní i (v RE notoricky známý) příznak IsDebuggerPresent na 1,
3. dle nastavení debuggeru jsou veškeré obsluhy výjimek[8] (exceptions) předány debuggeru – tzn., jakmile dojde v procesu k výjimce, není tato chyba předána obsluze programu, ale debuggeru, který upozorní uživatele.

Debugger pak může běh programu ovládat prakticky libovolným způsobem, od standardních příkazů typu step forward (krok dopředu), step-out atd., může samozřejmě používat i tzv. breakpointy. Ty fungují pro nás velice zajímavým způsobem – do paměti na místo instrukce, na které chceme běh programu zastavit, debugger zapíše instrukci INT 3 (interrupt 3, opcode CC), která vyvolá výjimku *EXCEPTION_BREAKPOINT*, která je následně předána k vyřízení debuggeru – ten jen zastaví běh programu.

Pokud se na celý princip fungování debuggeru podíváme ještě jednou s odstupem, bez pochyb vypozorujeme mnoho způsobů, jak přítomnost debuggeru detekovat. Oficiálních metod, jak debugger detekovat není mnoho, ale triků (tzv. antidebug triky) založených na detailních znalostech fungování debuggerů a jejich efektu v systému a různých dalších triků existuje několik desítek. Způsoby detekce debuggeru můžeme rozdělit na několik skupin:

- oficiální – pomocí IsDebuggerPresent,
- debugger princip – záměrné vyvolávání chyb v programu (pokud chyba neskončí v obsluze programu, je program debuggován), testování různých příznaků (IsDebuggerPresent, HEAP flag stacku, …), hledání instrukce INT 3 zapsané v kritických funkcích programu, …,
- trikové – měření času v různých kritických součástech programu, kdy několika vteřinová prodleva může znamenat jen zvědavý pohled analytika.

[7] Funkce poskytované samotným systémem, většinou skrze interface DLL knihoven typu kernel32, user32 atd.

[8] Tzv. SEHy – Structured Exception Handling, v mnohých jazycích známé jako Try-Catch-Exception.

Z výše zmíněných způsobů může být jistě patrné, že záležitost detekce debuggeru je do jisté míry věcí fantazie, a zároveň platí, že čím rozmanitější a neobyčejnější taková detekce je, tím hůře je odhalitelná pro případného útočníka. Nyní se podívejme na nejznámější zástupce debuggerů:

OllyDbg (Oleh Yuschuk)
Tento debugger je mezi crackery a celou scénou reverzního inženýrství znám pro svou kompaktnost, funkčnost a efektivitu. Tento nástroj bez pochyb stojí za velkým množstvím vydaných cracků, keygenů, analýz virů všeho druhu a dalších tématických subjektů.

WinDbg (Microsoft)
Jedná se o velmi mocný a po technické stránce propracovaný debugger od Microsoftu, se kterým je možné ladit jak jádro, tak běžné programy. Rozhodně stojí za zmínku i jeho schopnost analyzovat chybové výpisy vytvářené Windows po modrých smrtích (BSOD). Bohužel není příliš uživatelsky přívětivý a práce s ním je vhodná spíše skrze příkazovou řádku, než uživatelskou interakcí s GUI.

Syser (SyserSoft)
Také velice mocný debugger, ale víceméně určený pouze pro debuggování jádra a ovladačů, než normálních programů. Svou podstatou pokračuje ve šlépějích svého univerzálního předchůdce SoftIce, ovšem již s uspokojivým uživatelsky přívětivým rozhraním.

2.2.3. Monitory

Zajímavým a také velice rozšířeným druhem utilit pro reverzní inženýrství jsou různé monitorovací aplikace. Ve srovnání s předchozími ryze programátorskými nástroji, nevyžadují monitory většinou jakékoliv hlubší znalosti programovaní nebo reverzního inženýrství, což má sice své výhody, ale na úkor všestrannosti a preciznosti použití, viz dále.

Účelem monitorovacích aplikací je pouze monitorování určitých konkrétních API funkcí nebo *tématických* skupin – např. všechny funkce pro práci se soubory, registry, procesy, sítí atd. Výsledkem takovéhoto monitoringu je pak detailní výpis použití těchto funkcí programem s jejich parametry atp. Monitory však všeobecně neobsahují funkce pro ovládání běhu programu tak, jak je známe u debuggerů – tato funkčnost je jim bezvýhradně ponechána. Funkčnost samotného monitorovacího softwaru lze sice snadno suplovat přímo samotným debuggerem pomocí breakpointů na začátcích funkcí, ale v případě, že chceme pouze odposlechnout volání některých funkcí nebo tématických skupin (někdy až desítky funkcí), může být debugger až příliš složitým a zbytečným – v takovýchto případech je výhodnější použít právě monitor.

Monitorovací programy mohou fungovat mnoha způsoby, ovšem všechny mají společný základ, kterým je tzv. *hákování* (od slova *hooking*) – tento pojem pro nás bude v dalším textu velice důležitým, proto je nutné si o něm vytvořit dobrou představu. Hákování můžeme jednoduše přirovnat k přesměrování toku dat skrze naši novou vloženou entitu. Klasickým příkladem hákování je přeřezání telefonního kabelu, mezi který se vloží telefonní sluchátko a kabel se zase propojí – veškerá další komunikace po tomto kabelu bude odposlouchávána stylem man-in-the-middle. Přitom naše vložená entita nehraje pouze pasivní roli, jakou je pouhý odposlech, ale disponuje mnohem větším potenciálem – např. „protékající“ data libovolně upravovat, filtrovat, uzavírat spojení atd. – zde záleží pouze na povaze mechanismu, přes který tato data tečou. Tento mechanismus je pak skrze jeho mnohostranný potenciál nazýván jako *handler* (obsluha, obsluhovač, manažer – také do budoucna velice důležitý pojem), protože je pouze věcí této obsluhy, co s nově příchozími daty udělá. Velmi detailně se budeme hákováním a jeho mechanismy zabývat v další kapitole. Monitorovací programy tedy jednoduše zaháknou potřebná místa a handlerem přeposílají informace o volání a stavu programu zpět monitoru. Podívejme se tedy na nejznámější zástupce různých monitorů:

Process Monitor (Microsoft)
Velice známý monitor umožňující sledovat pouze *tématické* skupiny – práci se soubory, registry, procesy a sítí. Monitorovat je možné jak celý systém, tak jednotlivé procesy. Nevyžaduje jakékoliv hlubší znalosti problematiky, což se samozřejmě odráží v jeho jednoduchosti.

API Monitor (Rohitab)
Jde víceméně o značně rozšířenou verzi Process Monitoru, kdy je možné monitorovat konkrétní API funkce nebo *tématické* skupiny.

WireShark (Wireshark.org), WPE Pro
Monitor veškeré síťové komunikace. Nutné podotknout, že je někdy výhodnější použít kompaktnějších služeb cíleného monitoru a editoru paketů WPE Pro.

Desítky dalších zástupců není třeba zmiňovat. Všechny programy v každé ze zmíněných kategorii spojuje stejná myšlenka a stejný koncept fungování. V této chvíli je nutné zmínit, že jsme prakticky vyčerpali všechny (!!!) nástroje „první linie“ reverzního inženýrství – přitom disassembler (statická analýza) a debugger (dynamická analýza) jsou ve většině případů jedinými spolehlivými nástroji pro precizní analýzu neznámého programu – díky své úplné kontrole nad ním. Naopak monitorovací programy typu Process Monitor nemají nad analyzovaným programem kontrolu prakticky žádnou (pouze jej „sledují“), a zároveň se také musí spoléhat na to, že zkoumaným programem nebudou odhaleny a ten

tak chytře nezmění své chování[9]. Měnit své chování může samozřejmě i program „pod“ debuggerem, ale to pouze v případě, že mu to neuváženým spuštěním (byť jen přeskočením jediné instrukce call) analytik dovolí. Je nutné si tedy uvědomit, jak velká propast je mezi precizní analýzou pomocí disassembleru + debuggeru a výsledkem monitorovacího software.

2.2.4. Unpackery, memory editory, dumpery, ...

Na začátku této kapitoly jsem zmiňoval důležitost faktu, že kód = data. Tato ekvivalence nám dává možnost chovat se ke kódu programu stejně, jako k jakýmkoliv jiným datům – tzn. kód programu můžeme komprimovat, šifrovat, rozsekávat na části apod. Právě šifrování, komprese a různé zpřehazování bloků kódu programů je jedním ze svatých grálů ochrany programů proti reverznímu inženýrství [6]. Pokud načteme do disassembleru program zašifrovaný tzv. PE cryptorem (PE šifrér, nebo jen cryptor / šifrér / packer), jediný kód který uvidíme, je speciální zavaděč, který tam byl vložen právě PE cryptorem proto, aby dešifroval kód aplikace a pak jej spustil (nebo ještě lépe, aby dešifroval jen další důležitý blok bezpodmínečně potřebný pro běh aplikace). V této chvíli existuje několik málo možností, co si se zašifrovaným programem počít:

- První možností je nastudovat v disassembleru, jak dešifrovací kód (zavaděč) funguje – což bývá docela obtížné samo o sobě, natož pak, když je takový kód znepřehledněn různými (ve výsledku) nic nedělajícími instrukcemi – tzv. anti-dasm makry. Pokud k takovému dešifrovacímu zavaděči přidáme ještě vlastnost, že je sám rozdělen do několika nezávislých bloků, které postupně dešifrují sami sebe, pak je statická analýza disassemblerem jednou z posledních věcí, kterou byste chtěli dělat (takovýto druh šifrérů je zcela běžný).

  Příklad použití anti-dasm maker:

```
Originál:                    Použití jmp-maker:
01: push  ebp                (1.) push  05
02: mov   eax, 123           (2.) ret
03:                          (5.) mov   eax, 123  !
04:                          (6.) jmp   07
05:                          (3.) push  ebp       !
06:                          (4.) jmp   04
```

Jak je vidět, makra kód naprosto znepřehlední.

[9]Vzhledem k množství virů, které jsou denně publikovány, již dávno antivirové společnosti rezignovaly na jejich precizní analýzu a většina virů je tak analyzována automaty na bázi různých monitorů a genetických analýz. Precizně analyzovány jsou jen viry něčím zajímavé.

Je nutné si uvědomit, že dešifrér (většinou) či dekompresor není záležitostí pár instrukcí, ale několika desítek až stovek instrukcí – přitom mezi každou z nich je umístěn skok na tu další, která je umístěna v paměti zcela náhodně atd. Tato skoková makra jsou ovšem jen jedním z mnoha rozličných způsobů, jak znepříjemnit analýzu kódu. Např., pokud se podíváme blíže třeba na instrukci `mov eax, 123`, tak tato instrukce by se dala bez pochyb přepsat na desítky jiných instrukcí, které by měly ekvivalentní výsledek a nulový vedlejší efekt[10].

- Druhou možností je pak krokování kódu zavaděče v debuggeru. Ovšem, pokud si představíme kód pro dešifraci datového bloku, kdy je každá instrukce přepsána na její ekvivalentní kombinaci více instrukcí, a to vše je ještě zabalené do neustálých skoků, zjistíme, že jediná *nová* výhoda debuggeru je v možnosti držet klávesu pro krok dopředu a sledovat, jak se kód krásně dešifruje. Ovšem toto dovolit by byla fatální chyba ochrany – proto jsou tato anti-dasm makra protkána navíc i triky na detekci debuggeru, které nekončí na zprávě typu „Byl jsi odhalen!“, ale v nekonečných extrémně složitých smyčkách vyúsťujících v Halting problém.

- Třetí možností je pak použít tzv. unpacker – nástroj přímo určený pro odstranění šifrérů a kompresorů z progamu. Tyto nástroje fungují relativně jednoduše – spustí chráněný program, počkají, až se celý načte (úspěšně proběhne dešifrace/dekomprimace), a následně uloží obsah operační paměti obsahující načtený program na disk (tzv. je *dumpnou* – dumping, specializovaný nástroj dumper). Na disku pak pouze upraví některé náležitosti a je hotovo. Hlavním problémem je otázka, kdy je vlastně program plně dešifrován a kdy je tedy nejvhodnější jej dumpnout – na tuto otázku se nedá univerzálně odpovědět a právě proto jsou používány přímo cílené unpackery na konkrétní PE šifréry a jejich konkrétní verze, založené na jejich specifickém chování.

Dalšími hojně používaným nástroji jsou různé PE editory/analyzátory zaměřující se na práci se spustitelnými soubory – např. na zobrazení všech PE struktur s možností je editovat; rozpoznání, zda je program chráněn a čím atd. Posledními nástroji jsou memory editory – editory operační paměti, se kterými je možné libovolně upravovat a spravovat paměť cizího procesu.

[10]Na přepisování jednotlivých instrukcí posloupností jiných instrukcí s ekvivalentním výsledkem jsou založeny mnohé polymorfní enginy virů.

3. Provádíme RE

Nyní již máme základní představu o tom, co reverzní inženýrství programů je a pomocí jakých nástrojů a proč se provádí. Podívejme se tedy na několik čistě hypotetických scénářů, které mohou nastat, a jaké mají důsledky. Jako základní premisu si udejme zcela precizně popsat fungování nového neznámého viru, od kterého máme spustitelný soubor.

3.1. Scénář č. 1 – Ideál

Nejprve začneme s nejjednodušší možnou variantou – nechráněným programem. Jeho zběžný popis by mohl vypadat následovně:

Kód:	soubor není šifrován, není použit žádný packer ani šifrér.
Jazyk:	jsou použity strojové instrukce procesoru (Assembler, jazyk C, ...).
Anti-debug:	žádné použité triky.
Anti-dasm:	žádné použité triky.

V tomto případě je možné program okamžitě nahrát do disassembleru a začít jej postupně analyzovat. Nebrání nám v tom žádné anti-dasm triky (jako jsou různá znepřehledňující makra), ani zašifrování kódu šifrérem. V případě, že je kód složitý, je možné jej bez problémů analyzovat *za běhu* pomocí krokování v debuggeru a doplňovat tak do disassembleru konkrétní hodnoty napovídající účel samotné funkce, proměnných atp., viz ukázka:

```
const void *sub_4010B0(a1,a2,a3){
 ...
 v5 = a1;
 v4 = malloc(0x1000u);
 v3 = (char *)v4 + 4096;
 for ( *((_BYTE *)v4 + 4096) = 0; v5; *(_BYTE *)v3 = v6 + 48)
 {
  if ( v3 < v4 ) break;
  v7 = v5;
  v8 = v5;
  v3 = (char *)v3 - 1;
  v5 /= a2;
  v6 = v8 - a2 * v7 / a2;
  if ( (unsigned int)v6 > 9 ) v6 += 7;
 }
 memcpy(v4, v3, v4 - v3 + 4097);
 *(_DWORD *)a3 = sub_40148D(v4, v4 - v3 + 4097);
 return v3;}
```

Obrázek 1. Analyzovaná funkce skrze HexRays

Kód je sice znovu rekonstruovaný do pseudo-jazyka blízkého jazyku C, ale i tak nám to dvakrát nepomáhá[11]. V takovémto případě je nejrychlejší použít debugger a podívat se, jaký vstup a výstup taková funkce má. Konkrétní vstupy této funkce zjištěné debuggerem jsou například: `integer 0x00123456`, `integer 16` a ukazatel na alokovanou paměť. Výstupem funkce (tj. proměnná `v3`) je ukazatel na textový řetězec „`123456`“. Od pohledu to vypadá, že funkce konvertuje zadané číslo (1193046.) do zadaného základu (16) a následně do textového řetězce. V této chvíli není nic jednoduššího, než funkci znovu zavolat s přepsanými parametry a experimentálně ověřit výstup. Pokud není možné, nebo je přespříliš složité (nebo nebezpečné) se dostat k volání takové funkce, je možné ji celou zkopírovat do vlastního programu a tam ji otestovat – to je sice snadné u malých funkcí, ale méně už u složitých a rozvětvených funkcí. Kopírování kódu je také závislé na prostředí, se kterým kód pracuje – pokud funkce pracuje s globálními proměnnými (ať už jde o uživatelské proměnné, nebo struktury objektově orientovaného jazyka), začíná být kopírování kódu skoro nemyslitelným, nebo složitějším, než jeho zkoumání v debuggeru.

Tento scénář je tedy jakýmsi ideálním stavem pro analýzu kteréhokoliv programu – pouze čistý kód bez ochran a pastí. Ovšem s takovýmto scénářem se u většiny dnešních virů lze setkat jen zřídkakdy – právě k němu se reverzní inženýři snaží dostat. To hlavní, co nám ale tento scénář ilustruje je, jak výhodné a komfortní je mít možnost program analyzovat pomocí disassembleru a v případě potřeby ověřovat nebo získávat informace pomocí debuggeru.

3.2. Scénář č. 2 – Anti-dasm

První scénář byl tím nejideálnějším – ale jaké problémy přinese, když je použit nový šifrér, který (čistě hypoteticky) znemožňuje dumpnutí (viz kapitola 2.2.4.) a zároveň však nejsou použity (nebo jsou požity pouze slabé a snadno odhalitelné) anti-debug triky?

Kód:	soubor je šifrován novým inteligentním šifrérem zabraňujícím dumpnutí.
Jazyk:	strojové instrukce procesoru (Assembler, jazyk C, …).
Anti-debug:	žádné, nebo snadno odhalitelné a snadno odstranitelné.
Anti-dasm:	žádné použité triky po dešifraci.

Tento scénář je velmi blízký realitě – co to však přesně znamená pro analytika? Analytik má, jak již bylo řečeno, několik málo možností – snažit se pochopit nový šifrér / zkusit najít již hotový unpacker, aby mohl používat disassembler a měl tak při analýze „pevnou půdu pod nohama“, nebo používat pouze debugger, což může být velmi nepříjemné – protože to, co dělá disassembler tak

[11] Typický zástupce kódu, kdy i ve zdrojovém textu s poznámkami tápete, co se vlastně a proč děje.

nenahraditelným, je způsob práce s ním – akademické pitvání s velmi snadným zapisováním výsledků, což zrovna není parketa právě debuggerů.

Tento scénář je svým charakterem (omezením pouze na debugger) velmi blízký situaci, kdy je kód a veškeré textové řetězce šifrován a k dešifraci dochází pouze při potřebě daný kód, nebo řetězec použít. V těchto případech pohled do disassembleru mnoho neprozrazuje a je nutná především práce v debuggeru, se kterým je nutné odhalit, co jednotlivé části a šifrované řetězce znamenají (což je velmi zdlouhavá a únavná práce).

Pokud jde tedy o závěr, který vyplývá z tohoto scénáře, pak je to jedině další zdůraznění, že disassembler a debugger jsou extrémně důležité nástroje. Co se ovšem stane ve chvíli, kdy je není možné použít na kód programu?

3.3. Scénář č. 3 – Interpretované jazyky

Jakmile analytik ztrácí možnost použít debugger a disassembler, ztrácí tím veškerý kontakt s reálným kódem programu – ztrácí schopnost kód a celý program precizně analyzovat. Tato situace nastává velmi snadno (a často) v případě interpretovaných jazyků vyžadující pro svůj běh interpret. Platforma .NET sice používá interpret a vlastní instrukce, ale význam těchto instrukcí je znám a přímo pro platformu .NET existují již hotové a funkční zpětné překladače zpět do zdrojového textu[12]. Proto je pro nás v tuto chvíli, a pro tento scénář, platforma .NET určitým způsobem nezajímavá. Větším problémem jsou z tohoto pohledu speciální ochrany, které přímo pro cílový chráněný program vytvoří vlastní unikátní interpret a původní kód takového programu pak přeloží do nových – interpretovaných – instrukcí. Výsledkem je tak program disponující jediným strojovým kódem, kterým je samotný interpret. Zbytek kódu programu je „zašifrován“ do neznámých instrukcí interpretu.

Analýza programu chráněného právě přepisem do neznámých instrukcí je velice obtížným úkolem. Na běžné interpretované jazyky (.NET, Java, VB PCode) existují disassemblery i debuggery, se kterými je možné provádět relativně plnohodnotnou analýzu, ale k unikátním a privátním (neveřejně distribuovaným) ochranám tohoto typu existují jen doporučení, jak je analyzovat. Přitom, existuje stále ještě jedna šance, která může v analýze pomoci – pokud ochrana nepoužívá (nebo používá slabé a snadno odhalitelné) anti-debug triky, je možné použít debugger na sledování a kontrolu používání API funkcí programem a používat memory breakpointy (breakpoint při přístupu k určité paměti, např. k textovému řetězci atp.) – krom analýzy samotného interpretu jsou to jediné možnosti, které nám debugger v tomto případě nabízí – ale i ty pro alespoň minimální analýzu stačí.

[12]Jako ochrana před tímto zpětným překladem interpretovaných jazyků se používají tzv. obfuskátory, které se snaží maskovat a přepisovat veškeré dostupné informace uložené ve spustitelném souboru.

Kód:	instrukce s neznámým významem.
Jazyk:	interpretovaný.
Anti-debug:	žádné, nebo snadno odhalitelné a snadno odstranitelné.
Anti-dasm:	instrukce s neznámým významem.

Není příliš časté, aby takovéto složité ochrany neměly propracovaný systém detekcí přítomnosti debuggeru. Tedy, pokud takto upravíme scénář na pravou míru, vzniká situace, kdy není možné použít ani debugger a ani disassembler. Jak se lidově říká: „a jsme nahraní“. Jediné nástroje, které nám mohou cokoliv o programu nyní říci jsou různé monitory (viz kapitola 2.2.3.), včetně speciálně upravených virtuálních strojů tak, aby vytvářely logy toho, co se v nich děje[13].

3.4. Výsledek

Pokud se ohlédneme na předchozí scénáře, je jistě patrné, že přijít o možnost použití debuggeru a disassembleru je velice snadné a vzhledem k jejich důležitosti i nesmírně komplikující. Tato situace vytváří potřebu nového nástroje, který by dokázal co nejblíže suplovat funkce debuggeru, a který by vyplnil mezeru mezi bezuzdnou nekontrolovatelností běžícího programu sledovaného monitorem a absolutní kontrolou debuggeru.

[13]Ke všem dostupným virtualizačním nástrojům jsou známé triky, jak jejich přítomnost detekovat. Od triků s posunutím LTD ukazatele, až po detekci nesprávně simulovaných instrukcí virtuálním procesorem aj.

4. Řešení chybějící mezery

Abychom mohli přemýšlet nad tím, jak tuto mezeru vyplnit, musíme nejprve vědět, čeho přesně chceme dosáhnout – musíme nejprve určit alespoň minimální funkčnost, která by vytvářela jakýkoliv krok od monitorů směrem k debuggeru a zároveň zůstala co nejméně detekovatelnou. Otázkou tedy je, co tak markantně odlišuje debugger od monitoru. Odpovědí je samozřejmě skutečnost, že běh programu „pod debuggerem“ je pod úplnou kontrolou uživatele – pomocí debuggeru může uživatel kdykoliv pozastavit běh programu na základě vnitřního zpracování kódu (breakpointy), vnitřních stavů nebo na základě jeho vlastní vůle. Právě tato schopnost pozastavit běh programu za určitých podmínek je tou nejdůležitější – pouze v pozastaveném stavu je možné bezpečně a efektivně přepisovat kód programu v paměti, upravovat hodnoty proměnných atd. Právě této schopnosti – dokázat běh programu pozastavit obdobně, jako v debuggeru – budeme chtít docílit za každou cenu.

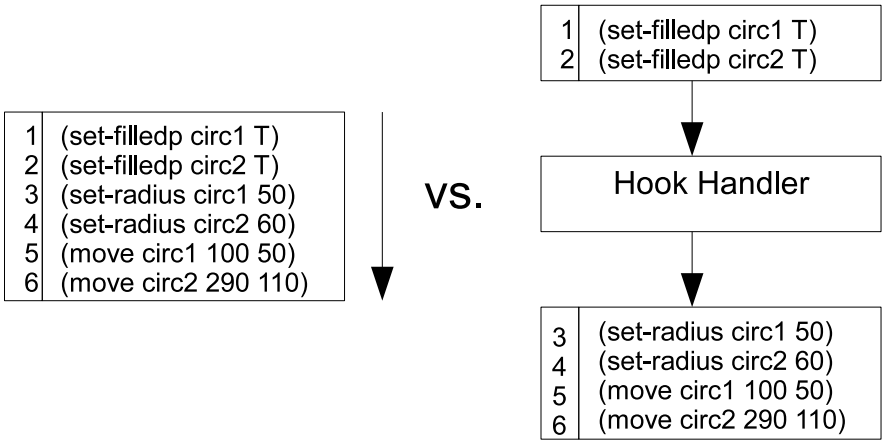
Uspokojujícím řešením vyplnění mezery mezi debuggerem a monitory je vize klasického API monitoru, který dokáže při volání zadané API funkce pozastavit běh programu a dovolit tak uživateli editovat vstupní parametry této funkce i jakoukoliv další paměť procesu, včetně kontextu vlákna (registry a flagy procesoru) a následně i editovat návratovou hodnotu této funkce. Jako maximum této vize se zdá být funkčnost monitorovat nejen API funkce, ale i libovolnou zadanou část kódu programu.

5. Použité metody

Cíle byly specifikovány, nyní již nezbývá nic jiného, než se je pokusit zrealizovat. Nejprve si upřesníme představy metod, které máme.

5.1. Hákování

S tímto pojmem jsme se již setkali v kapitole věnované samotným Monitorům, a vytvořili si o něm určitou představu, kterou si nyní raději připomeňme: jde o vložení naší určité entity (handleru) mezi dva komunikující body. Takové dva body mohou být ovšem i v programu a je jen na nás, co za tyto dva body prohlásíme. Na obrázku 2. můžete vidět, jak vypadá hákování v kontextu programu.



Obrázek 2. Originální a zaháknutý kód.

Zcela naivním řešením hákování je jednoduše vložit celý kódu handleru mezi ty řádky kódu, které spolu s jejich kontextem (prostředím) chceme nějakým způsobem kontrolovat (ovládat, mít schopnost s nimi libovolně manipulovat). Ovšem pokud si uvědomíme, že nepracujeme se zdrojovým textem, ale pouze s binárními daty – strojovým kódem, jehož instrukce se na sebe různě odkazují a jejich jednoduché „posunutí níže“ jako u textu nepřichází v úvahu, nastává problém, jak tedy do kódu „vecpat“ náš celý handler (případně jen jeho volání), který bude simulovat právě činnost breakpointů.

Pokud si vzpomeneme na princip fungování debuggerů, tak si uvědomíme, že debugger také používá metodu hákování. Debuggery jednoduše přepíší instrukci kódu programu tam, kam chce uživatel nastavit breakpoint, na instrukci INT 3 (která má 1 byte – tedy nejmenší možná instrukce). Procesor při zpracování této instrukce vyvolá výjimku (chybu) typu *EXCEPTION_BREAKPOINT*, a protože

debugger přebírá (přesměrovává – hákuje) veškeré zpracování (řízení) výjimek debuggovaného programu do svého vlastního handleru, je pak pouze jeho věcí, jak výjimku zpracuje – běh programu může bez problémů pozastavit (– uživatelský breakpoint). Debugger pak následně pouze přepíše nazpět přepsaný první 1 byte původní instrukce, která je pak po opětovném spuštění ihned provedena. Problém s přesunováním kódu tedy debugger neřeší – zapisuje prakticky pouze volání handleru. Přitom toto volání má délku pouhý 1 byte (!) – každá instrukce má minimální délku 1 byte a tudíž je vždy přepsána instrukcí INT 3 právě maximálně jediná instrukce, která je následně „opravena“ a zpracována. Tato metoda je tou nejjednodušší a nejefektivnější, bohužel je však zcela závislá na k tomuto účelu specializované instrukci přerušení INT 3 a přebírání řízení výjimek nad celým debuggovaným programem – což je pro naše účely nepoužitelné, protože se chceme jakýmkoliv (nebo alespoň většině) detekcí běžných debuggerů vyhnout.

Pokud si shrneme, na čem obecně staví funkčnost breakpointů v klasickém debuggeru, pak jde o 1. volání handleru (pomocí hardwarově podporovaného přerušení INT 3) a 2. zpracování tohoto volání handlerem (pozastavení programu). Jediná reálná možnost, jak kód programu donutit zpracovat náš handler a nepoužít přerušení, je zkrátka přepsat instrukce programu na volání / skok do našeho handleru (tzv. inline hákování, s dalšími typy hákování se setkáme později).

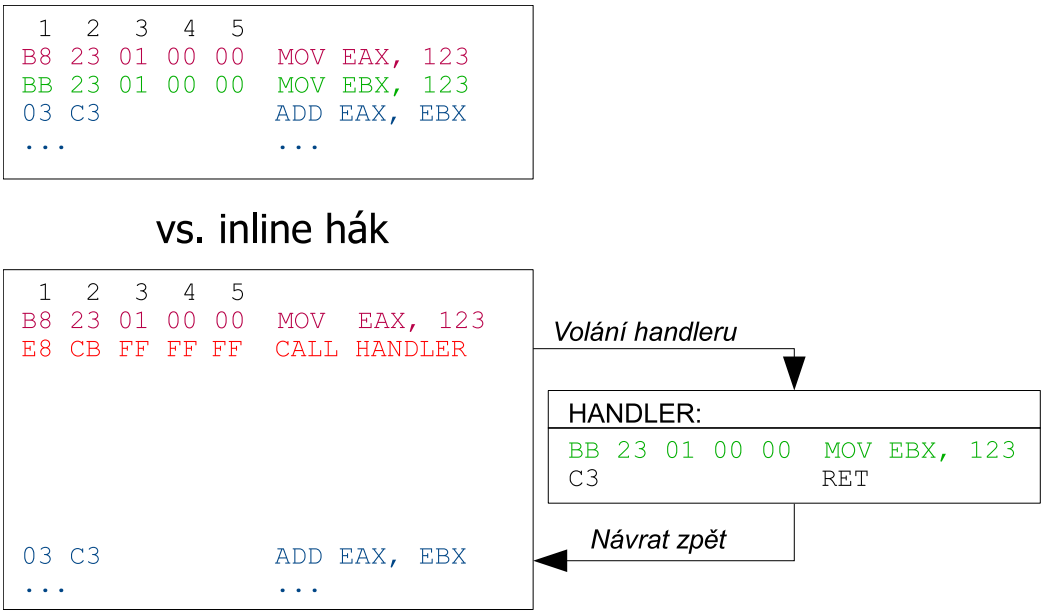
5.2. Inline hákování – koncept breakpointu

Pokud se pokusíme chovat obdobně jako debugger, narazíme na jeden zajímavý problém. Pokud budeme přepisovat jednotlivé instrukce programu na skok do našeho handleru (realizující „pseudobreakpoint“) a pak je zase pro pokračování obnovovat na jejich původní hodnoty, vystavujeme se riziku, že v době mezi obnovením původních instrukcí, jejich zpracováním a znovu nastavením skoku (breakpointu), mohou další vlákna přes právě dočasně odstraněný breakpoint procházet, což je fatální chyba. V debuggeru při použití klasických debug API jsou všechna vlákna při breakpointu pozastavena, což tento problém zcela jednoduše eliminuje, ale zároveň vytváří jiný – co když nechceme nebo dokonce nesmíme pozastavit všechna vlákna? Například, jakmile se připneme debuggerem k procesu subsystému Windows – procesu CSRSS – celý systém zamrzne, což je právě způsobeno pozastavením všech životně důležitých vláken subsystému – přitom pozastavení jednoho určitého vlákna by nebyl problém.

Protože chceme vytvořit i co nejpoužitelnější náhražku za debugger a monitor zároveň, budeme počítat i s možností, že bude uživatel chtít zkoumat i kritické procesy typu CSRSS. Což nás vede k nutnosti vymyslet jiný koncept, než při breakpointu a následném spuštění přepisovat přepsané instrukce nazpět (protože tohle nevyhnutelně vede k nutnosti pozastavení všech vláken).

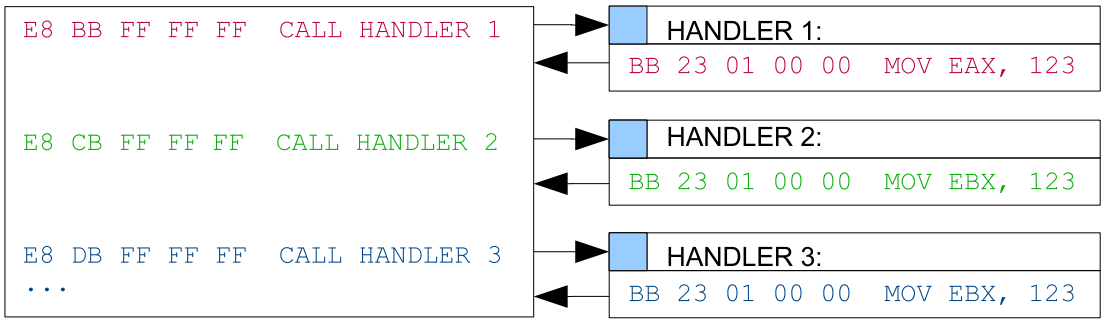
Řešením je přesunutí přepsaných instrukcí přímo do handleru, což je relativně blízko k „posunování“ instrukcí, ale s tím rozdílem, že maximální počet přepsaných instrukcí je roven délce volací / skokové instrukce (při použití instrukce

CALL / JMP jde tedy maximálně o 5 přepsaných instrukcí). Tím, že prakticky zálohujeme přepsané instrukce do handleru, se vyhneme několika nutnostem – zapisovat přepsané instrukce zpět, pozastavovat všechna vlákna a pak vláknu, které breakpoint spustilo, znovu nastavit počáteční adresu instrukcí, které má provést. Konkrétní ukázka inline háku vedoucího do handleru, který provede přepsané instrukce a vrátí se zpět, vypadá následovně (Obrázek 3.):



Obrázek 3. Originální a zaháknutá funkce inline hákem.

Pokud do handleru před vykonání přepsané instrukce `MOV EBX, 123` vložíme ještě zobrazení modálního MessageBoxu, pak máme prakticky fungující breakpoint, kdy vlákno nepokračuje dál, dokud není MessageBox uživatelem odklepnut. Vzhledem k tomu, že handler obsahuje přepsané instrukce té části kódu, která byla přepsána na volání / skok na tento handler, je zcela důležité si uvědomit, že každý hák (každý breakpoint) znamená právě jeden handler, viz Obrázek 4.:



Obrázek 4. Zaháknuté všechny řádky kódu.

5.2.1. Volání handleru

Na výběr, jak náš handler volat mnoho možností nemáme, tedy pokud chceme být ekonomičtí[14].

Instrukce JMP a CALL: jediný rozdíl mezi těmito dvěma instrukcemi je v tom, že CALL přidává na vrchol zásobníku adresu zpět, kterou pak odebírá a na ni se vrací instrukce RET. Vzhledem k tomu, že pro každý hák tak jako tak musíme mít vlastní handler (obsahující specificky pro dané místo zálohované přepsané instrukce), nepotřebujeme CALLem *pushnutou* návratovou adresu, protože tu po zapsání háku známe (viz dále), a na konec háku tak můžeme vložit přímo skok zpět. CALL je pro nás tedy zbytečně složitý.

Instrukce JMP *adresa* je dlouhá 5 bytů [4]. Při zapisování této instrukce může dojít k několika situacím:

1. Délka první přepisované instrukce je rovna délce instrukce JMP:
   situace např. z Obrázku 3., kdy JMP i přepisovaná instrukce jsou stejně dlouhé. Návratová adresa je pak jednoduše *Adresa JMP + 5*. Zálohovaná instrukce je pouze jedna.

2. Délka první instrukce je větší než délka instrukce JMP:
   V tomto případě se instrukce JMP „vleze“ přímo do přepisované instrukce o délce n, ze které zbyde $n - \textit{délka JMP}$ bytů nepřepsaných. Jelikož je zálohována celá instrukce, je návratová adresa rovna *Adresa JMP + n*.

   Např.: `[67 68 44 33 22] 11 - push 11223344`, byty v hranatých závorkách budou přepsány instrukcí JMP; byte `11` zbyde a sám o sobě vytvoří instrukci `OR`, tento byte tedy musí být přeskočen. Zálohovaná instrukce je pouze jedna.

3. Délka první instrukce je menší než délka instrukce JMP:
   Aby nebyl kód poškozen, je nutné zálohovat všechny přepsané instrukce – proto je nutné zjistit, kolik instrukcí (a jakých) bude přepsáno: v jednoduchém cyklu stačí procházet následující instrukce a sčítat jejich délky do doby, než délky těchto instrukcí $m \geq$ délce instrukce JMP – všechny tyto instrukce pak musí být zálohovány.

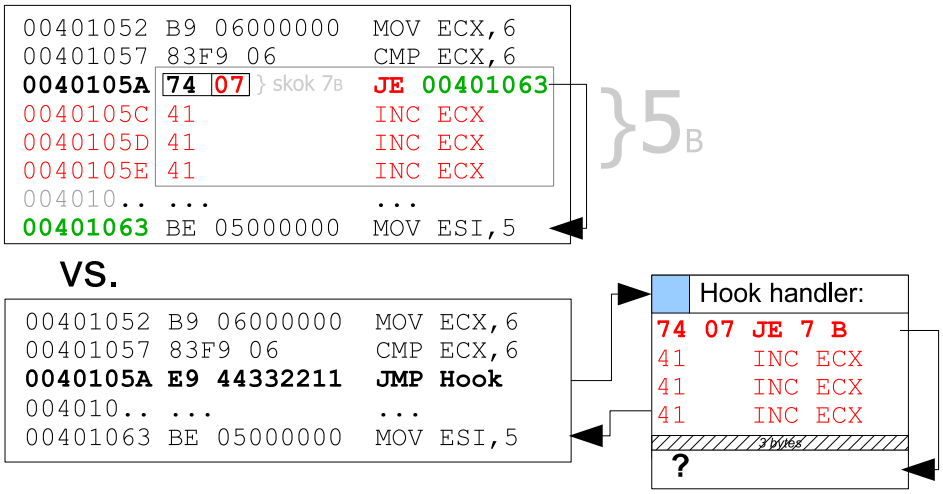
   Např.:

```
[55             push ebp        (1)
 8B EC          mov  ebp, esp   (1+2)
 83 EC] 3C      sub  esp, 3C    (1+2+3 = 6 = m), m >= 5 OK
 56             push esi        <- návr. adresa
...
```

[14] Ve smyslu co nejvíce ušetřit.

Byty v hranaté závorce budou přepsány na instrukci JMP. Zálohovány však budou všechny instrukce, kterým byl přepsán kterýkoliv byte (analogicky k bodu 2.), tedy: PUSH EBP, MOV EBP, ESP, SUB ESP, 3C, a návratová adresa je rovna *Adresa JMP + délka přepsaných instrukcí m.*

Zálohování – přesun přepsaných instrukcí na jiné paměťové místo – přináší jeden fatální problém, pokud zůstane neřešen, a tím je znevalidnění relativity instrukcí. Mnoho instrukcí, jako skoky, podmíněné skoky, ale i volání jsou relativní – instrukce nemá uloženou přímou adresu, kam bude skákat, ale má uložený pouze počet bytů o kolik skočí dopředu / dozadu (počítáno od následující instrukce za skokem). Blízké podmíněné skoky, jako `JZ - 74 xx` a `JNZ - 75 xx` tak zabírají pouhé dva byty, přičemž druhý byte právě specifikuje počet bytů skoku dopředu / dozadu. Pokud ovšem „vytrhneme“ tuto instrukci ze svého kontextu, bude skákat vzhledem k nové adrese naprosto špatně (což je zřejmé, viz Obrázek 5.):



Obrázek 5. Skok do neznáma.

Relativní podmíněné skoky

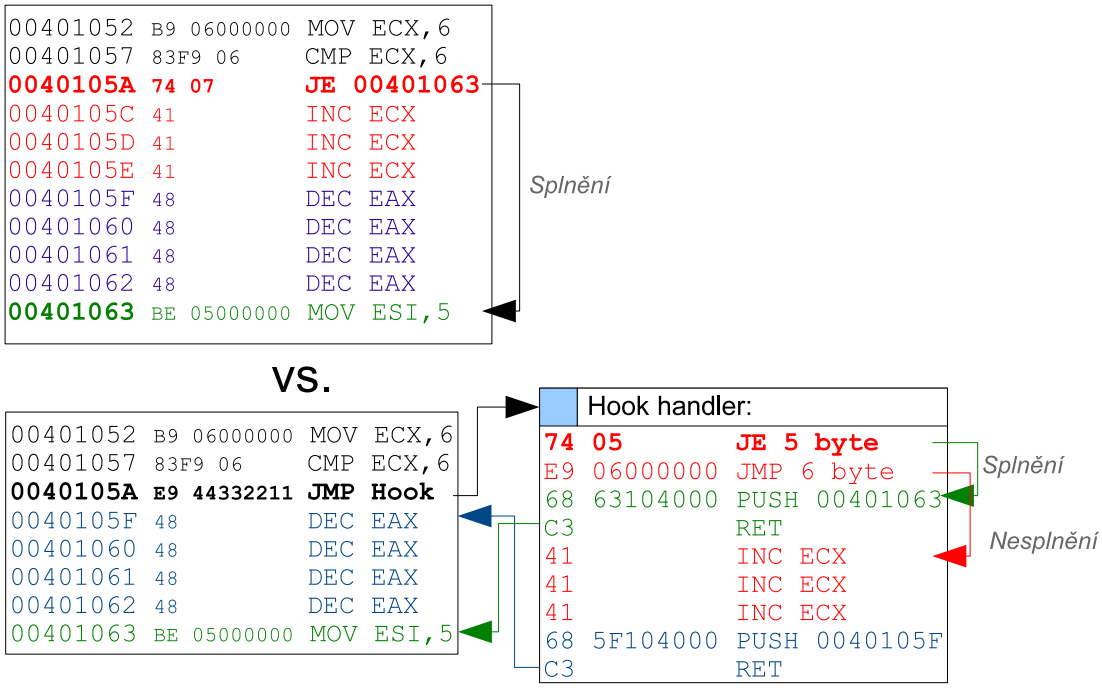
Řešením tohoto problému je 1. detekovat, že se jedná o relativní instrukci a 2. použít trik, pomocí které upravíme relativní skok na absolutní, a to prakticky bez jakékoliv práce a komplikací. Podívejme se zpět na Obrázek 5. – co znamená splnění podmínky? Skok *naslAdresa JE*[15] + *počet bytů*, tj. 00401063. Nesplnění podmínky znamená pokračovat následující instrukcí – a právě část z těchto následujících instrukcí bude spolu s podmínkou zálohována do handleru (zvýrazněné červeně). Zbytek instrukcí, které následují dále, zůstávají na místě – tedy na adrese *Adresa JE + délka přepsaných instrukcí*, tj. 0040105F – a zde je pak nutné pokračovat.

```
74 09       - JE 7 bytů dopředu
41          - INC ECX
41          - INC ECX
41          - INC ECX
E9 98541200 - JMP 0040105F (PUSH & RET)
```

Na místo relativního skoku JMP (který vyžaduje dopředné kalkulace založené na umístění zdroje a cíle) je výhodnější použít dvojici instrukcí `PUSH` & RET – PUSH vloží na vrchol zásobníku adresu cíle, tj. 0040105F, a RET na ni skočí. (JMP ani CALL nemají jednoduché absolutní varianty.) Skok zpět při nesplnění podmínky již máme vyřešen. Pokud však podmínka bude splněna, pak skok o 7 bytů dopředu v našem handleru není validní – validní je samozřejmě pouze v původním umístění této instrukce a ne v našem handleru. My víme, že podmíněný skok skáče při splnění podmínky na adresu 00401063. Otázkou však je, jak donutit vyzálohovanou instrukci podmíněného skoku, aby na tuto adresu skočila. Okamžitá myšlenka – přepsat vzdálenost skoku instrukce na nově vykalkulovanou, okamžitě naráží na fyzické omezení velikosti operandu nesoucí právě tuto vzdálenost (pouze 1 byte pro blízké skoky). Další obdobné myšlenky končí na příliš velké složitosti způsobené např. množstvím podmíněných skoků a jejich unikátními formulemi splnitelnosti atp.

Řešením je velice jednoduchá metoda: v původní instrukci podmíněného skoku změníme vzdálenost, o kterou skáče, vždy na 5 bytů dopředu – takto upravená instrukce pak skočí přímo na absolutní skok PUSH & RET vedoucí na původní adresu splnění podmínky (00401063). Oněch 5 bytů, které upravená instrukce skáče, stačí na přeskočení nepodmíněného skoku, který skáče na tělo nesplněné podmínky přes PUSH & RET (6 bytů) (Obrázek 6.):

[15] Protože relativní skoky počítají vzdálenost od svého konce, je nutné použít značení: naslAdresa značí následující adresu za skokem a Adresa absolutní adresu instrukce.



Obrázek 6. Originální a zaháknutý kód.

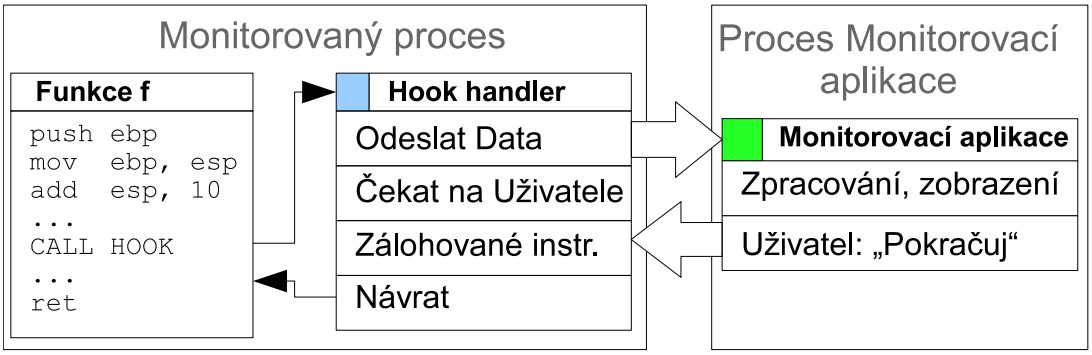
Jak je vidět na Obrázku 6., oba kódy jsou navzájem ekvivalentní. Relativní podmíněné skoky jsou tedy pomocí této metody „zabaleny“ do jakéhosi univerzálního makra, které je transformuje na absolutní, a tedy validní kdekoliv v paměti. Tento trik navíc funguje pro všechny podmíněné skoky stejně (jen posunutě vzhledem k samotné délce instrukcí skoku). Nyní ještě zbývá vyřešit triviálnější problém s nepodmíněnými relativními skoky.

Relativní nepodmíněné skoky
Jedná se pouze o instrukce CALL a JMP. Vzhledem k tomu, že skáčí vždy, není problém je přepsat na absolutní skok na předem vypočítanou adresu (adresa + počet bytů). U instrukce CALL ještě před finálním PUSH RET musíme *PUSHnout* adresu následující instrukce po instrukci RET (tj. *adresa PUSH* + *délka PUSH* + *délka RET*).

5.2.2. Handler a breakpointy

Nejprve si vymezme několik pojmů. *Monitorovací aplikací* myslíme naši cílovou aplikaci, která bude operovat s monitorovaným programem. *Monitorovaným programem* myslíme monitorovaný/debuggovaný program naší Monitorovací aplikací. *Breaknutým stavem* myslíme stav, ve kterém je vlákno monitorovaného programu pozastaveno a čeká na spuštění Monitorovací aplikací.

Realizaci skoku na handler již máme, včetně extrémně důležité správné zálohy přepsaných instrukcí. Nyní je nutné se zamyslet, co vše bude handler dělat – nezapomeňme, že cílem je vytvořit náhradu za debugger i monitor zároveň. Nejzákladnější funkcí handleru tak bude odeslat informaci o spuštění breakpointu (*triggernutí*) naší Monitorovací aplikaci. Druhou funkcí, nezbytnou pro správné fungování breakpointů, je následné vyčkání, dokud uživatel znovu nespustí běh monitorovaného programu (z breaknutého – suspendnutého stavu).



Obrázek 7. Komunikace monitorovaného programu s Monitorovací aplikací.

Komunikace na základě Obrázku 7. musí obnášet: prvotní signál o spuštění breakpointu (zpracovávání handleru) monitorovaným programem a následné čekání, dokud uživatel v Monitorovací aplikaci znovu nedovolí programu pokračovat. Tyto požadavky na komunikaci vypadají nevyhnutelně na kombinaci eventů, semaforů a trubek, avšak existuje již hotové řešení, které nám ušetří spousty práce – klasické okenní zprávy Windows Messaging (WM) a WM API. WM API po odeslání zprávy čekají, dokud nedostanou od adresáta odpověď. Vzhledem k tomu, že můžeme jednoduše u svého okna přidat libovolný handler těchto zpráv (WindowProc, v C++ jako WndProc), máme vyřešenou komunikaci a hlavně, dokud vlákno neopustí funkci WindowProc / WndProc, vlákno v monitorovaném programu bude čekat na místě – breakpoint vyřešen, přitom handler bude volat jedinou API – SendMessage [5]:

```
DWORD SendMessage lib "user32.dll" (DWORD hwnd, DWORD wMsg, _
                                    DWORD wParam, DWORD lParam)
```

Jak je vidět v deklaraci: parametr `hWnd` je unikátní identifikátor cílového okna (adresáta), `wMsg` je zpráva typu WM, přičemž můžeme použít i vlastní hodnotu, která identifikuje, že právě přijatá zpráva je od handleru v monitorovaném programu. `wParam` a `lParam` jsou parametry závislé na typu zprávy (`wMsg`), což pro nás znamená dva libovolné parametry, kterými můžeme předat libovolné 32 bitové hodnoty. Nejvýhodnější využití těchto dvou parametrů je pro Identifikátor Procesu (PID) a adresu paměti monitorovaného programu, na které začínají důležitá data, viz dále.

Důležitá data, která by měla být uživateli předána:

1. ID vlákna – identifikace vlákna, které spustilo breakpoint,
2. registry procesoru – všeobecně důležité, jednotlivé registry procesoru a jejich hodnoty, především pak registr EAX používaný pro návratovou hodnotu funkcí, ECX a EDX používané pro předávání parametrů atd.,
3. flagy procesoru – jednotlivé příznaky (zero-bit, carry bit, ...) potřebné pro zjištění, zda bude podmíněný skok skákat, pro měnění logiky těchto skoků atp.,
4. ukazatel na zásobník – kvůli zjišťování parametrů, návratovým adresám, které zde ukládají instrukce CALL, buffery atd.

Pokud si uvědomíme, že všechna tato data dávají smysl pouze tehdy, jsou-li v autentickém – tedy námi jakkoliv nezměněném – stavu (ať už se jedná o registry, flagy nebo zásobník), pak se zákonitě musíme chovat při psaní handleru velice obezřetně ke všem použitým instrukcím, a to do takové míry, aby neměnily vůbec nic, nebo aby bylo možné kteroukoliv z takových změn vrátit do původního autentického stavu. Jinak neriskujeme pouze odeslání zkreslených dat Monitorovací aplikaci, ale i pád samotného monitorovaného programu, jemuž měníme data „pod rukama“. Tím největším nebezpečím je zde právě volání Win API SendMessage, která dle konvence stdcall může měnit kterékoliv registry a flagy, kromě ESI a EDI. Je tedy nutné vytvořit zálohu celého kontextu, který bude po dokončení veškeré komunikace znovu obnoven – ovšem před vytvořením zálohy a po jejím obnovení musí být vše do nejmenších detailů promyšleno. Správně napsaný hák se tedy tváří tak, jakoby ani neexistoval. Otázkou je, zda je tento požadavek pro nás nějakým problémem. Odpověď zní – ne, není, a dokonce nám zjednoduší a ušetří mnoho práce. Proč?

Abychom zjednodušili komplexnost samotného handleru, vytvoříme pomocnou funkci, která bude zaštiťovat kompletní komunikaci pomocí API SendMessage – tuto funkci pojmenujme jako CallHandler, tedy:

Program (hák) $\Rightarrow$ *handler* $\Rightarrow$ *CallHandler (komunikace)*

Právě v této funkci bude soustředěné veškeré zálohování procesorového kontextu, protože ve zbytku kódu (tj. handler) dokážeme provést všechny potřebné operace čistě bez změny byť jediného flagu.

Ve funkci CallHandler využijeme instrukce `pushad` a `pushfd`. Instrukce `pushad` zálohuje na zásobník všechny registry (na zásobník přidá 9 DWORDů v pořadí: EDI, ESI, EBP, ...), instrukce `pushfd` vloží na zásobník 1 DWORD obsahující pomocí bitové masky uložené všechny flagy procesoru. Na zásobníku je tedy přidaných 10 DWORD hodnot. Pomocí instrukcí `popfd` a `popad` jsou tyto vyzálohované flagy a registry ze zásobníku vyjmuty a znovu načteny.

Příklad použití instrukcí `pushad` a `pushfd`:

```
start: eax == 100, zeroflag == 0, esp == 0
pushad
pushfd
  // esp == 40 ((9 + 1) * 4 byte)
  mov eax, 123  // eax = 123
  sub eax, 123  // eax = 0, zeroflag == 1
popfd
popad

konec: eax == 100, zeroflag == 0, esp == 0
```

Mezi těmito dvěma instrukcemi můžeme dělat prakticky cokoliv (samozřejmě kromě bezduchého přepisování zásobníku). Právě zde můžeme bez problémů zavolat naši funkci `SendMessage (hWnd Monitoru, WM Breakpoint, PID, ESP)` a předtím ještě na zásobník vložit i ID aktuálního vlákna:

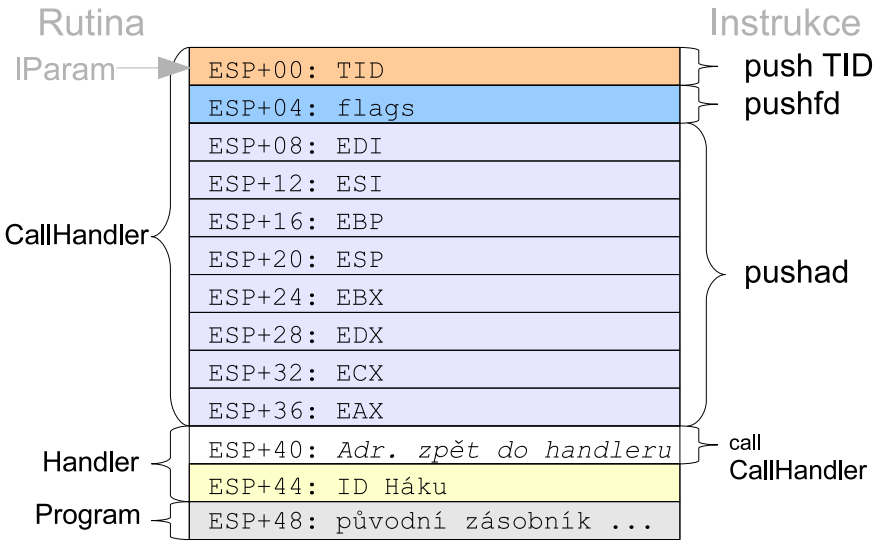
```
proc CallHandler:
pushad
pushfd
  -- vstup do kritické sekce  -------------------

  push TID            //ID vlákna
  push ESP            //ESP ukazuje na hodnotu TID, pod kterou jsou
                      //flagy, pod kterými jsou registry, viz obr.
  push PID            //ID procesu
  push WM_Breakpoint
  push hWndMonitor
  call SendMessage

  add  esp, 4         //odstranění přidaného TID

  -- výstup z kritické sekce  -------------------
popfd
popad
ret
```

Monitorovací aplikace tedy získá zprávu, ve které parametr lParam ukazuje přímo na začátek všech důležitých dat v monitorovaném programu (procesu) (Obrázek 8.). Ty jsou navíc přepisovatelné – takže pokud chce uživatel změnit hodnoty registrů nebo flagů, pak lze jednoduše přepsat[16] právě ty hodnoty, ze kterých budou registry a flagy procesoru po spuštění obnoveny (pomocí `popad`, `popfd`).



Obrázek 8. Přesný stav zásobníku po `push TID`

Vzhledem k tomu, že funkce SendMessage čeká, dokud adresovaný proces nezpracuje WM zprávu, je zásobník vzhledem k adrese předané v lParam neměnný. Monitorovací aplikace tak může s registry, flagy a celým zbytkem zásobníku pracovat dle libosti a spoléhat na offsety (vzdálenosti vůči lParam) znázorněných v Obrázku 8.

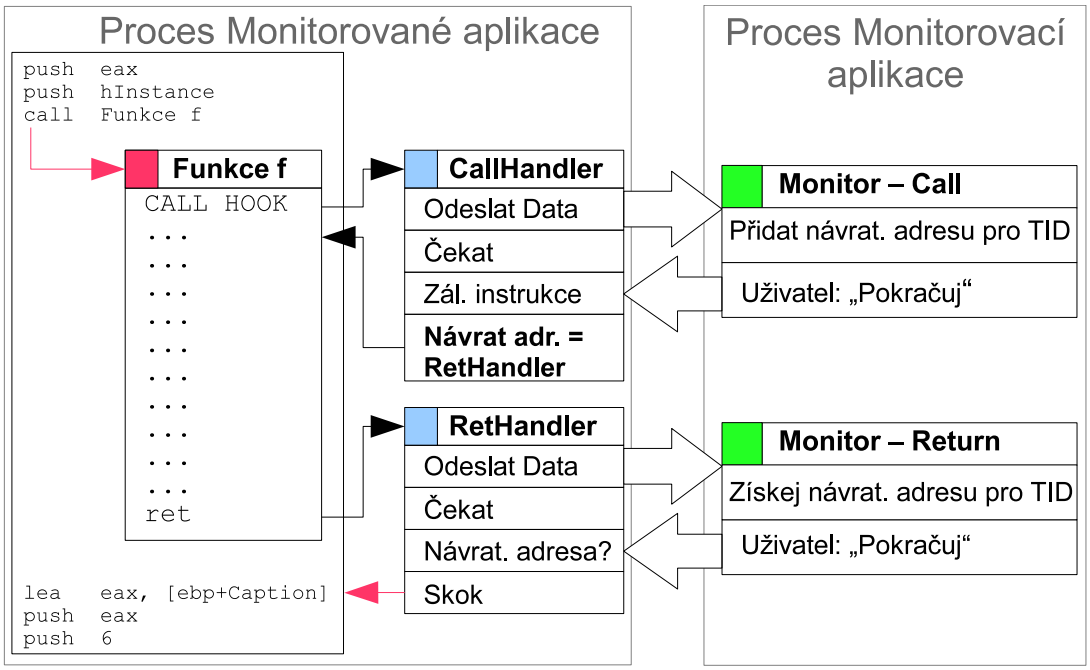
5.2.3. Získávání návratových hodnot funkcí

Původní zásobník (offset ≥ 48) jsou data, která program uložil do zásobníku před skokem do handleru – pokud zahákneme **začátek** libovolné funkce, pak je přímo na offsetu 48 návratová adresa, kam se funkce po jejím dokončení vrátí (tzn. návratová adresa). Pokud přepíšeme návratovou adresu na adresu našeho dalšího „návratového“ handleru, který bude informovat Monitorovací aplikaci, pak budeme mít k dispozici jak data z volání určité funkce, tak i data z jejího návratu (návratová hodnota, stav paměti, ...), a to včetně možnosti pozastavit běh programu / vlákna před zpracováním funkce a následně i před návratem funkce do programu.

[16] Prací s pamětí a procesem celkově se budeme věnovat v samostatné kapitole.

Jak již bylo řečeno, toto platí pouze v případě, pokud je zaháknut začátek funkce – tedy v jediném případě, kdy je jisté, že první hodnotou v zásobníku je návratová adresa – v jiných případech nemá smysl přepisovat neznámou hodnotu na adresu ReturnHandleru! Proto je nutné začít rozlišovat účel breakpointu – zda jde o „monitorovací breakpoint“, nebo „debuggerovský breakpoint“ – právě tyto dva typy breakpointů vyjadřují dualismus Monitorovací aplikace zastupující jak klasické API monitory (hákující začátky funkcí a tudíž i s potřebou znát návratovou hodnotu skrze níže popsaný postup), tak debuggery (sledující pouze aktuální stav na zadané instrukci).

Samotná implementace monitorovacích breakpointů (tedy těch, které vyžadují přepsání návratové adresy) je následující: handlery jsou zapisovány ve dvojici – jeden zpracovávající volání a druhý návrat (ReturnHandler). Přepsání návratové adresy musí logicky proběhnout v handleru při volání. Zde však vzniká velký problém: jakmile je funkce dokončena a vrátí se na ReturnHandler (skrze přepsanou návratovou adresu), nemá již nikde zapsanou původní návratovou adresu určující, kam se vrátit zpět do programu. Návratovou adresu však není možné uložit do jednoduché globální proměnné kteréhokoliv z handlerů z jednoduchých důvodů: 1) pokud se monitorovaná funkce volá rekurzivně, pak prvním rekurzivním voláním je návratová adresa přepsána návratovou adresou právě do této funkce – což nevyhnutelně vede k implementaci zásobníkového typu ukládání návratových adres řešící tento problém a 2) analogicky k prvnímu bodu se přidává ještě možnost, že je monitorovaná funkce volána více vlákny zároveň – takže pro každé jedno vlákno musí existovat jeden suplující zásobník návratových adres identifikovaný pomocí Unikátního identifikátoru vlákna (TID). Samotná implementace dynamických zásobníků identifikovaných pomocí TID přímo v handleru je příliš složitá a zcela zbytečná – výhodnějším řešením je přesunout kompletní suplování zásobníků návratových adres do Monitorovací aplikace. Monitorovací aplikace pro každé nově monitorované vlákno (tedy to, které spustí breakpoint, identifikované pomocí unikátního TID předaného na offsetu 0), vytvoří zásobník, do kterého bude vkládat návratové adresy. Jakmile bude zpracováván ReturnHandler, Monitorovací aplikace vyjme první hodnotu zásobníku daného vlákna a pomocí návratové hodnoty `SendMessage` (která je rovna návratové hodnotě funkce WindowProc / WndProc) předá původní návratovou adresu, na kterou ReturnHandler skočí – program tedy bez problémů pokračuje. Následuje kompletní schéma monitorovacího breakpointu (Obrázek 9.):



Obrázek 9. Zaháknutá funkce, zpracování volání a návratu.

Obrázek 9. přesně ilustruje fungování monitorovacího breakpointu, pouhý *debuggovací breakpoint* má svou analogickou ilustraci na obrázku 7.

5.2.4. Rekapitulace

Realizace našeho breakpointu je založena na inline hákování – tedy přepsání n instrukcí monitorovaného programu na jedinou instrukci skoku do handleru. Handler následně volá komunikační funkci, která odesílá Monitorovací aplikaci veškerá důležitá data a vyčkává na odpověď. Handler pak na základě toho, zda jde o *monitorovací*, nebo *debuggovací breakpoint* přepíše návratovou adresu na ReturnHandler, který informuje Monitorovací aplikaci o návratu z monitorované funkce a provede správně zálohované instrukce. Breakpoint je plně funkční bez jakýchkoliv hardwarově podporovaných přerušení a přebírání řízení výjimek či čehokoliv jiného.

5.3. Realizace a implementace breakpointu

Nyní jsme ve fázi, kdy víme, čeho chceme dosáhnout a proč – zatím však nevíme jak. Jak vůbec zapisovat do cizího procesu? Jak v CallHandleru vůbec volat API funkci? A co teprve pak, když v monitorovaném procesu není nahraná ani knihovna user32.dll, která tuto funkci exportuje? Jaké nástrahy skýtá psaní handleru, který nesmí změnit jediný flag? Vše bude detailně vysvětleno právě v této kapitole.

5.3.1. Proces a spustitelné soubory

V této kapitolce si vytvoříme představu o pojmu proces – tuto představu budu vytvářet pouze na základě těch informací, které v našem kontextu mají nějaký význam a tedy nás neohrožují zbytečným zabředáváním do – pro nás – nepodstatných témat.

Spustitelným souborem rozumíme soubor typu EXE, ale jsou to prakticky i soubory typu DLL, OCX, SYS a další. Každý z těchto souborů je uložen ve speciálním strukturovaném formátů pojmenovaném jako Portable Executable Format, zkráceně PE Format – ten obsahuje všechny důležité informace nutné ke spuštění a správnému fungování programu (např. adresa kódu, na kterém má program začít – tzv. EntryPoint, atd.).

Co se tedy děje při spuštění EXE souboru je, že systém pomocí PE Loaderu načte EXE soubor do paměti, vytvoří nové vlákno na adresu specifikovanou ve struktuře PE – tedy na EntryPoint a program běží. Toto zjednodušené schéma nám postačí. Nejdůležitější částí, která nás v tomto procesu zajímá, je vyplnění tabulky importů PE loaderem.

Tabulka importů (Import Allocation Table, dále jako IAT) je uložena v PE struktuře souboru. IAT obsahuje seznam knihoven a jejich funkcí, které spustitelný soubor používá („importuje“) – tyto knihovny jsou postupně načteny a adresy funkcí jsou do IAT tabulky doplněny. Spustitelný soubor se pak při volání těchto funkcí odkazuje právě na tabulku importů[17]. Pokud takto načítaná dll knihovna sama o sobě importuje (využívá) jiné knihovny, jsou PE Loaderem postupně rekurzivně načteny také. Doplnění adres jednotlivých funkcí probíhá následovně:

1. Je načtena PE struktura vyžádaného dll souboru,
2. z této struktury je načtena Tabulka Exportů (EAT – Export Alloc. Table), která obsahuje informace o všech exportovaných funkcích,
3. v tabulce je požadovaná funkce nalezena buď podle jména (tj. např. „MessageBox“), nebo dle číselného identifikátoru (zastaralé).

[17] Např., volání funkce MessageBox v C vytvoří volání CALL DWORD PTR[IAT:MessageBox], tedy „volat uložený dword na adrese IAT položka MessageBox“)

Každý proces obsahuje po svém vytvoření minimálně dva moduly[18] – soubor, kterým byl proces vytvořen (např. EXE) a knihovnu ntdll.dll. Knihovna Ntdll je důležitou fyzickou bariérou-bránou mezi uživatelským režimem (ring 3) a režimem jádra (ring 0). Právě skrze tuto knihovnu jsou volány funkce jádra běžným programem a knihovnami. Funkce exportované Ntdll – tzv. Zw API – pouze „přeposílají“ volání skrze bránu (sysenter / int 2e) do ekvivalentních NT funkcí v jádru systému (exportovaných ntoskrnl/kpa) – pro zjednodušení budeme označovat i Zw API jako NT. Právě skrze ntdll probíhají veškeré práce se soubory, registry, objekty, procesy atd. Ale jelikož je ntdll pouhým „prostředníkem“ mezi jádrem a uživatelským režimem, pak je ntdll pouhým zrcadlením verze jádra a jeho funkcí – což je z hlediska kompatibility problém, který je řešen abstrakčními bariérami – knihovnami jako kernel32, advapi32 a další. Právě z těchto důvodů, aby uživatelé používali abstraktně oddělené funkce exportované knihovnami kernel32 atd., Microsoft oficiálně nedokumentuje žádnou z NT API funkcí (a pokud dokumentuje, pak s velkým upozorněním, že neručí za jakoukoliv změnu)[19].

Nejdůležitější je tedy představa, že fyzicky běžící program reprezentují moduly načtené do paměti (exe, dll, ...). V procesu bude vždy načten minimálně spouštěný exe soubor a ntdll. Jakékoliv další moduly jsou načteny pouze na základě toho, že je vyžaduje – uvádí ve své tabulce importů (IAT) – exe soubor (ntdll ji absolutním minimem, tedy nemá žádný import) a následně rekurzivně načítané moduly.

Př.) EXE soubor využívá pouze funkci SendMessage exportovanou z knihovny user32.dll:

Načtení modulu	IAT modulu	Načtené moduly
EXE soubor	user32 (SendMessage)	EXE, ntdll, user32
user32	ntdll, kernel32, gdi32	EXE, ntdll, user32, kernel32, gdi32
kernel32	ntdll	EXE, ntdll, user32, kernel32, gdi32
gdi32	ntdll, kernel32, user32	EXE, ntdll, user32, kernel32, gdi32

Tabulka 1. Posloupnost načítání modulů.

[18]Modul = libovolný PE loaderem načtený PE soubor.

[19]Neoficiální dokumentace těchto API jsou tvořeny pomocí reverzního inženýrství.

5.3.2. Základní práce s procesy

Nyní se zaměřme na základy práce s procesy. Abychom mohli pracovat s většinou objektů ve Windows (a procesy nevyjímaje), je nutné je nejprve „otevřít" – získat handle. Handle (česky popisovač / rukojeť), je objekt popisující typ přístupu k danému objektu, práva přístupu a mnoho dalších informace o vztahu vytvářející entity a cílové entity. Např. pokud chceme pracovat s cizím procesem a známe jeho identifikátor (PID), pak použijeme funkci `OpenProcess`, ve které specifikujeme typ přístupu k procesu, který chceme k *otevíranému* procesu mít, a funkce nám na základě systémových práv vrátí buď handle, nebo chybu. Analogicky je to s vlákny (`OpenThread`), systémovými registry a dalšími objekty. Otevřený handle zůstává v paměti systému do doby, než je uzavřen pomocí univerzální funkce `CloseHandle`, nebo do ukončení procesu, který jej vytvořil.

Nyní se podívejme na výčet základních API funkcí pro práci s procesy, které v budoucnu využijeme:

Jméno	**Účel**	**Poznámka**
`CreateProcess`	Vytvoří nový proces ze zadaného PE souboru	Vrací ID, handle procesu a prim. vlákna
`OpenProcess (id)`	Získá handle procesu na základě jeho ID	
`CreateRemoteThread`	Vytvoří vlákno v zadaném procesu	Vrací handle vytvořeného vlákna
`OpenThread (id)`	Získá handle vlákna na základě jeho ID	
`ResumeThread (h)`	Uvolní zadané vlákno	
`SuspendThread (h)`	Pozastaví zadané vlákno	
`GetThreadContext (h)`	Získá kontext zadaného vlákna	
`SetThreadContext (h)`	Nastaví zadanému vláknu zadaný kontext	
`VirtualAllocEx (h)`	Alokuje v zadaném procesu paměť	Vrací ukazatel na alokovanou paměť
`VirtualFreeEx (h)`	Uvolní paměť v zadaném procesu	
`WriteProcessMemory (h)`	Zapíše do paměti zadaná data	
`ReadProcessMemory (h)`	Přečte z paměti zadaná data	
`TerminateProcess (h)`	Ukončí zadaný proces	

Tabulka 2. API funkce pro práci s procesy. Zdroj: [5]

5.3.3. Zápis vlastního kódu do cizího procesu

Do cizího procesu můžeme zapisovat prakticky kamkoliv a cokoliv, pokud na to ovšem máme v systému privilegia – v účtu adminstrátora máme práva ke všem procesům a na účtu hosta pouze k těm, které vytvoříme.

Ke vložení kódu do cizího procesu máme dvě možnost: vložit do procesu naši vlastní vytvořenou dll knihovnu, nebo zapisovat přímo jen čistý kód do námi alokované paměti. **Vlastní DLL knihovna** má výhodu v tom, že její vytvoření je prakticky bezpracné, jednotlivé funkce jsou z knihovny exportovány a tudíž je jednoduché se na ně odkazovat, knihovnu samotnou stačí do cílového procesu pouze „nainjektovat“ (násilně načíst; tuto metodu si ozřejmíme později). Nevýhodou této metody je, že je 1. jednoduše detekovatelná, 2. skrýt ji před detekcemi je neúměrně náročné, 3. pokud není napsaná přímo v assembleru, nemáme úplnou kontrolu nad výsledným kódem (což je nebezpečné, viz kapitola o psaní handleru). Oproti knihovně DLL je **zapisování čistého kódu** o něco náročnější na přípravu, ale několikrát se vynaložená práce v budoucnu vrátí. Vytvoření čistého kódu lze dosáhnout dvěma způsoby:

1. nechat kód zkompilovat spolu s aplikací, a pak jej dle potřeby v paměti před zapsáním upravovat (tzv. patchování). Tento postup je zcela běžný u hákujících aplikací napsaných v assembleru, ale neproveditelný v interpretovaných jazycích, nebo

2. napsat si vlastní malý překladač assemblerovských instrukcí, kterých je potřeba dohromady něco kolem šesti.

Vytvoření malého překladače assemblerovských instrukcí je univerzální a snad i nejrozumnější volbou pro kterýkoliv jazyk – výsledná implementace může nabrat podoby funkcí, které vrací strojový kód instrukcí v textovém řetězci jako hexdump (binární data v hexadecimální podobě), který je před zapsáním do paměti převeden do binární podoby. Toto řešení má největší výhodu v jednoduchosti použití:

```
string code;
asm x86 = new asm();

code = x86.Push(REG_ESP) +
       x86.Push(WM_BREAKPOINT) +
       x86.Call(pSendMessage);
```

Výsledkem je řetězec `code = "5466 689604 E8076FD376"`, který stačí převést na pole bytů a zapsat do paměti procesu pomocí `WriteProcessMemory`. (Jak již může být patrné, napsat Monitorovací aplikaci bude možné prakticky v jakémkoliv jazyce, ve kterém je možné pracovat s API funkcemi.)

5.3.4. Spouštění procesu

Abychom mohli efektivně monitorovat kterýkoliv program od jeho začátku, musí být už při jeho spuštění monitorován – tedy vše potřebné musí být už před jeho spuštěním zaháknuto. Co potřebujeme je, vytvořit proces, ale nespouštět jej – přímo k tomuto účelu slouží parametr `dwCreationFlags` ve funkci `CreateProcess` [5]:

```
CreateProcessA(
    LPCSTR lpApplicationName,
    LPSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCSTR lpCurrentDirectory,
    LPSTARTUPINFOA lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation);
```

Tato API funkce vytváří proces a pokud není specifikováno jinak, pak jej i spustí – parametr `dwCreationFlags` s hodnotou `CREATE_SUSPENDED` však vytvoří proces, vytvoří prvotní (primární) vlákno, ale to zůstává na EntryPointu v nečinném (suspendnutém) stavu – program tedy není spuštěn – což je přesně ten stav, který potřebujeme, abychom mohli zaháknout vše potřebné; pro spuštění programu pak stačí vlákno uvolnit (resumnout).

Je zde ovšem jeden zádrhel – nejen že takto vytvořený proces zůstává nespuštěný, ale vlákno stojící na EntryPointu – primární vlákno – je právě tím vláknem, které načítá tabulku importů (IAT), což ve zkratce znamená, že takto vytvořený proces obsahuje ve svém pozastaveném stavu pouze dva moduly: exe a ntdll. Pokud však chceme monitorovat například vytváření souborů skrze funkci `CreateFile` z kernel32, pak jednoduše nemůžeme – knihovna kernel32 obsahující tuto funkci ještě nebyla načtena, a analogicky je to i s ostatními funkcemi (kromě NT API).

Existují dvě řešení tohoto problému: pomocí `CreateRemoteThread` a pomocí `SetThreadContext`.

CreateRemoteThread

CreateRemoteThread vytváří vlákno do libovolného procesu. Inicializace samotného procesu (tedy vyplnění IAT a načítání modulů, ...) není čistě věcí primárního vlákna – tímto úkolem je pověřeno kterékoliv prvně spuštěné vlákno procesu. Jednoduše tedy stačí vytvořit nové vlákno např. na funkci `CsrGetProcessId()` v ntdll, a to následně inicializuje celý proces bez spuštění jeho kódu.

S touto metodou je však spjat jeden velký problém, který se konkrétně týká „načítání modulů“ a jejich inicializace námi vytvořeným remote vláknem. Po načtení dll knihovny do procesu je zavolána její inicializační rutina (`DllMain()`) a pokud z nějakého důvodu taková knihovna předpokládá, že ji vždy inicializuje primární vlákno a na tomto předpokladu staví veškeré (případně jen některé) své fungování, pak tu máme velký problém. Tento inicializační problém je zřejmý u knihovny user32, která je pro naše účely klíčová, a tedy nemůžeme riskovat její jakékoliv poškození, navíc nemůžeme vzhledem k zacílení Monitorovací aplikace na viry spoléhat, že ty se budou vždy chovat korektně (ať už s nějakým důvodem, či ne)[20].

SetThreadContext
Tato metoda je o něco složitější, ale na rozdíl od předchozí nevytváří žádné anomálie ani chyby. Spočívá v jednoduchém principu: pozastavené primární vlákno „chytneme“, přesuneme na náš vlastní kód vytvářející nekonečnou smyčku (spinlock, který do paměti procesu zapíšeme) a vlákno spustíme – proces bude inicializován a primární vlákno bude cyklit v nekonečné smyčce – původní program tak nebude spuštěn. Monitorovací aplikace pak na základě výčtu všech importů bude kontrolovat, zda jsou již v procesu načteny všechny potřebné moduly. Jakmile budou požadované moduly načteny, primární vlákno se pozastaví a přesune zpět na EntryPoint.

Přesunutí vlákna na jinou adresu je pouhou otázkou změny jediného registru kontextu vlákna, a to registru EIP (ten ukazuje na aktuální zpracovávanou instrukci) pomocí funkce `SetThreadContext`. Nejprve je však nutné do procesu zapsat kód spinlocku, ve kterém následně bude primární vlákno „uvězněno“ – pro tyto účely nám bohatě stačí např. následující kód:

```
_zpet: NOP
       PAUSE
       JMP _zpet
```

Kompletní postup pro správnou inicializaci nově vytvořeného procesu pomocí `CreateProcess` s flagem `CREATE_SUSPENDED` je následující:

1. `OpenThread (TID)` – handle primárního vlákna vráceného funkcí `CreateProcess` nemá správný typ přístupu – proto je nutné vytvořit nový handle pomocí OpenThread s flagy `THREAD_SET_CONTEXT` || `THREAD_GET_CONTEXT` ⇒ `hThread`.

2. `VirtualAllocEx (hProcess)` – alokujeme stránku paměti s ochranou `PAGE_EXECUTE_READWRITE`, aby bylo možné do ní zapisovat a spouštět v ní kód ⇒ `pCodePage`.

[20] Tato chyba, alias USER32 bug byla zdokumentována českým programátorem EliCZ.

3. `WriteProcessMemory (hProcess, pCodePage)` – zápis kódu spinlocku do alokované paměti.

4. `GetThreadContext (hThread)` – získání aktuálního kontextu vlákna ⇒ `tContext`, abychom mohli po dokončení čekání nastavit veškeré hodnoty nazpět (především pak původní EIP). Registr EIP v `tContext` se nastaví na `pCodePage`.

5. `SetThreadContext (hThread, tContext)` – vláknu je nastaven nový kontext, nyní vlákno stojí na začátku našeho spinlocku.

6. `ResumeThread (hThread)` – vlákno je uvolněno, proces je právě inicializován.

7. Čekání na načtení potřebných modulů (cyklický test na načtení všech importů EXE).

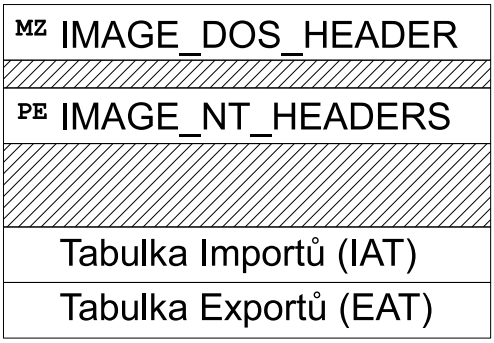
8. `SuspendThread (hThread)` – po načtení všech potřebných modulů pozastavíme primární vlákno a kontextu `tContext` nastavíme původní hodnotu EIP (předem zálohovanou),

9. `SetThreadContext (hThread, tContext)` – nastavíme zpět vláknu původní kontext.

V této chvíli je proces inicializován a připraven k hákování.

5.3.5. PE struktura, IAT, EAT a hákování

Z celého PE formátu nás nejvíce zajímají již několikrát zmíněné tabulky IAT a EAT. Pomocí nich můžeme určit, kdy je proces načten a zároveň je můžeme použít přímo i pro účely samotného monitorování. Nejprve si ale udělejme alespoň minimální představu, jak vypadá spustitelný soubor v PE formátu (Obrázek 10.) [3]:



Obrázek 10. Ukazatele na IAT a EAT struktury.

Handle modulu ⇒ `hModule` (který vrací např. funkce `LoadLibrary`) je pouze pseudohandle – ve skutečnosti se jedná pouze o pointer ukazující na začátek modulu v paměti procesu, který je oficiálně označen jako `ImageBase`. (Počáteční adresu modulu v paměti cizího procesu lze nejjednoduššeji získat výčtem všech modulů procesu pomocí funkcí `CreateToolhelp32Snapshot` s flagem `TH32CS_SNAPMODULE`.) Přímo na začátku PE souboru (tj. `ImageBase`) je DOS hlavička, tu využijeme pouze k vypočítání adresy PE hlavičky (souboru několika hlaviček alias NT Headers) obsahující ukazatele na IAT a EAT.

Tabulka Exportů (EAT)
Tabulka exportů by se dala velice jednoduše popsat jako dvojrozměrná tabulka obsahující veškeré veřejné-exportované funkce daného modulu a jejich počáteční adresy. Pro nalezení počáteční adresy funkce v modulu se používá funkce `GetProcAddress`, která nefunguje nijak jinak, než že iterativně prochází EAT zadaného modulu a porovnává jednotlivá jména s tím zadaným. Funkce `GetProcAddress` však funguje pouze lokálně – nelze ji použít pro nalezení adresy funkce v cizím procesu. Vzhledem k tomu, že v našem handleru budeme volat API funkci `SendMessage`, bude nutné nějakým způsobem adresu této funkce nalézt, a to nejlépe předem, ne až v handleru. Nejvýhodnější cestou je tak napsat vlastní mezi-procesní GetProcAddress, který bude procházet moduly a jejich EAT v cílovém procesu[21].

[21] Sice je možné použít způsob „stejných vzdáleností“, kdy funkce `SendMessage` bude ve stejné vzdálenosti vzhledem k počátku dll knihovny v paměti našeho procesu a i v cizím, ale jde pouze o předpoklad.

Další důležitou věcí u EAT je fakt, že obsahuje ukazatele na začátky jednotlivých funkcí – pokud tyto adresy zaměníme za ukazatel na náš handler, který pak originální funkci zavolá, máme další typ háku – **EAT hák** – druhý nejrozšířenější typ hákování. Ten je výhodný zapsat před spuštěním procesu, kdy další načtené moduly vyplní do svých IAT námi přepsané adresy. Tento typ háku funguje i pro mnoho programů napsaných v jazyce C, ve kterých mnoho programátorů používá pro volání API funkcí `LoadLibrary` a `GetProcAddress` za běhu (někdy i na každé volání).

Tabulka Importů (IAT)
Tabulka importů je podobná tabulce exportů, avšak je tu jedna změna – jelikož PE soubor může importovat různé funkce z více knihoven, tak se jedná o trojrozměrnou tabulku tvořenou seznamem knihoven, seznamem jejich importovaných funkcí a jejich počátečních adres, které PE loader získá právě z EAT těchto modulů. Tabulka importů existuje z jednoduchého důvodu – při načítání modulů do paměti není nikdy zaručeno (ve Windows 64bit je tomu tak schválně kvůli buffer-overflow exploitům), že tyto moduly budou vždy na stejné adrese + jakákoliv nová verze takové knihovny by mohla znamenat posunutí kódu v nich, a tedy i adres začátků jednotlivých funkcí – proto jsou jednotlivé adresy načítány dynamicky při spuštění / vyžádání. Program, resp. PE soubor, tedy po kompilaci neobsahuje u volání funkcí importovaných z dll knihoven jejich adresu, ale pouze odkaz do Tabulky importů:

```
Jazyk C:
Sleep (100);

Překlad:
push 100
call dword ptr[IAT:KERNEL32->SLEEP]
```

Instrukce call tedy volá funkci specifikovanou v políčku IAT pod knihovnou kernel32 se jménem SLEEP.

IAT sám o sobě je pro nás drahocennou pomůckou pro zjišťování, zda je proces již načten[22], ale také je dalším vhodným místem, které se dá velice efektivně díky svému konceptu hákovat – jde o tzv. **IAT hák** – prvním nejrozšířenějším a nejjednodušším typem hákování. Pokud zahákneme veškeré funkce uvedené v IAT programu (který dále nepoužívá žádná dynamická volání např. pomocí `GetProcAddress`), pak veškerá jeho volání půjdou skrze naši Monitorovací aplikaci. Problémem této metody je její jednoduchost, která z ní vytvořila notoricky známý prostředek špehování a většina softwaru, který chce cokoliv ukrýt, IAT pro tyto účely nepoužívá.

[22]Tato metoda popsaná v kapitolce 5.3.4. Spouštění procesu však není stoprocentní, protože moduly sice mohou být načtené v paměti, ale stále může probíhat jejich vnitřní inicializace, proto je vhodné ještě pár vteřin počkat.

5.3.6. Injektáž DLL knihoven do cizího procesu

Tato schopnost je pro nás rozhodující v případě, že monitorovaný program nijak neimportuje (ani skrze importy všech ostatních modulů) pro nás klíčově důležitou knihovnu user32 obsahující odesílací funkci `SendMessage`. Právě pomocí DLL injektáže však dokážeme do takového procesu tuto knihovnu bez problémů vložit a používat.

Koncept DLL injektáže tkví v zapsání jednoduchého kódu do cílového procesu, který zadanou knihovnu načte. Pro načtení knihovny do procesu se používá funkce `LoadLibrary`, jejíž parametr je cesta k dll knihovně.

```
push pZapsanaCesta          ;cesta k dll knihovně
call LoadLibrary            ;načtení knihovny
ret
```

Cesta ke knihovně je samozřejmě do procesu zapsána také. Nyní stačí pouze pomocí funkce `CreateRemoteThread` vytvořit vzdálené vlákno, které tento kód provede a vyčkat na jeho dokončení.

Avšak cílový program vůbec nemusí importovat ani knihovnu kernel32, která obsahuje právě funkci **LoadLibrary** – a bez knihovny kernel32 jsme v koncích – tedy pouze oficiálně. Funkce `LoadLibrary` funguje pouze tak (mimo pár dalších drobností), že svůj parametr – cestu k dll knihovně – překonvertuje do struktury `UNICODE_STRING` a zavolá funkci `LdrLoadDll` exportovanou z ntdll.dll, která je nahraná v každém procesu:

```
LdrLoadDll(
  IN PWCHAR               PathToFile OPTIONAL,
  IN ULONG                Flags OPTIONAL,
  IN PUNICODE_STRING      ModuleFileName,
  OUT PHANDLE             ModuleHandle );
```

Není tedy problém použít místo `LoadLibrary` funkci z ntdll `LdrLoadDll`, pomocí které můžeme nahrát i zmíněný kernel32.

5.4. Finální implementace handlerů

Nyní víme prakticky vše potřebné k tomu, abychom mohli vytvořit finální implementace jednotlivých handlerů. V této podkapitole se tedy seznámíme se všemi handlery, jejich řešeními a dokumentací.

5.4.1. CallHandler

Nejprve se podívejme na hlavní CallHandler – reference a původní koncept můžete nalézt v kapitole 5.2.2..

```
Hlavička:
mov  edi, edi
  PushAD                                    ;zachování registrů
  PushFD                                    ;zachování flagů

  ;-- critical section    --------------------
  push lpCriticalSection                    ;-|
  mov  eax, lpEnterCriticalSection          ; |-- (1)
  call eax                                  ;-|
```

Jak již bylo jednou řečeno – instrukce call nemá jednoduché absolutní volání, pokud tedy chceme volat cokoliv zadané absolutní adresou, pak pomocí triku **(1)**, kdy se do kteréhokoliv registru vloží pointer a registr se „zavolá“. Tento trik má hlavní nevýhodu v tom, že přepíše registr – což nám po vyzálohování všech registrů nevadí. Tímto způsobem je volána funkce `EnterCriticalSection`, která je důležitá pro zamezení zahlcení Monitorovací aplikace (pokud si představíme například hákování funkce `send` v prohlížeči Firefox, která je volána několika vlákny zároveň, pak by byla Monitorovací aplikace zahlcena zprávami a mohlo by dojít k problémům).

```
Tělo:
      push fs:[00000024]                    ;(2) PEB->CurrentTID (!+)

      push esp                              ;lParam: ESP
      push fs:[00000020]                    ;wParam: PEB->PID
      push WM_MONITOR_CALL_HANDLER          ;wmMsg:  BREAKPOINT WM
      push p_lSubClassFormHwnd              ;hWnd:   hwnd okna
      mov  eax, lpSendMessage
      call eax                              ;call SendMessage

      add  esp, 4                           ;(3) (!-)
```

Řádek označený **(2)** je *pushnutí* hodnoty s indexem 0x24 v TIB (segment Thread Information Block, ve kterém jsou uloženy důležité informace o aktuálním vláknu), tedy ID aktuálního vlákna (TID), nápodobně pak hodnota s indexem 0x20 – ID procesu (PID). Řádek **(3)** odstraňuje ze zásobníku *pushnutý* TID.

```
Konec:
  push lpCriticalSection
  mov  eax, lpLeaveCriticalSection
  call eax
  ;----------------------------------------

  PopFD
  PopAD
  ret
```

Volání odchodu z kritické sekce, obnova flagů a registrů, návrat do handleru.

5.4.2. Handler

Nyní je výhodné se vrátit k původnímu handleru – tedy tomu, do kterého vede hák a obsahuje přepsané instrukce s voláním CallHandleru. Připomeňme si, že právě tyto handlery jsou ty, ve kterých nesmí být změněn kterýkoliv flag a registr.

Monitor handler – handler volání, přepisující návratovou adresu na adresu return handleru:

```
CallHook handler:
  push lHookId                                  ;(!+)

  mov  dword ptr[esp-4], lpMonitorCallHandler  ;(1)
  call dword ptr[esp-4]                         ;call CallHandler

  pop  dword ptr[esp-4]                         ;(2) (!-)

  mov  dword ptr[esp], _pReturnHandler          ;(3)

  -- zde vložit opravené (!) přepsané instrukce;(4)
  ...
  --

  push (pHook + lRewroteInstructionsLength)     ;(5)
  ret
```

Prvním trikem, pomocí kterého voláme CallHandler **(1)**, je absolutní call, kdy adresa není uložena do registru, ale do zásobníku pod ESP, kde se nachází zatím nevyužitá paměť. Tento trik je možné obejít pomocí použití standardního relativního volání, ale bylo by nutné předem rozpočítávat, kde instrukce call bude, aby bylo možné vypočítat vzdálenost skoku. Proto je jednodušší a univerzálnější použít absolutní skok. Trik č. **(2)** je zde kvůli odstranění vloženého ID háku ze zásobníku (nutného pro identifikaci háku Monitorovací aplikací) – tato situace se standardně řeší pomocí instrukce add, ale ta mění flagy procesoru, což je pro nás naprosto nepřípustné[23]. Tento trik přesune ze zásobníku první položku a vloží ji do nepoužité části. Řádek **(3)** přepisuje vrchol zásobníku, na kterém je uložena návratová adresa – ta je přepsána na ukazatel na ReturnHandler. **(4)** je místo pro zálohování instrukcí, po jejich provedení je proveden absolutní skok zpět **(5)** do kódu programu.

```
ReturnHook Handler:
  push lHookId                                  ;(1) (!+)

  push pMonitorReturnHandler                    ;(2)
  ret
```

(1) zase identifikuje hák pro Monitorovací aplikaci a následně je proveden skok na odeslání dat ReturnHandlerem.

[23]Např. když budeme hákovat instrukci podmíněného skoku, pak můžeme neuváženým měněním flagů měnit kompletní chování celého monitorovaného programu.

Breakpoint handler
BreakpointHandler je prakticky totožný s Monitorovacím breakpointem – jediným rozdílem je však chybějící řádek **(3)**, který přepisuje vrchol zásobníku na adresu ReturnHandleru.

5.4.3. ReturnHandler

ReturnHandler, tedy analogická verze CallHandleru určená k umožnění monitorování návratu z funkce je teoreticky popsána na straně 35. Důležité je si připomenout, že návratovou hodnotou námi volané funkce `SendMessage` je adresa, na kterou se po dokončení handlování má program vrátit.

```
mov edi, edi
PushAD
PushFD

;-- critical section    --------------------
  push tCS.lpCriticalSection
  mov eax, tCS.lpEnterCriticalSection
  call eax

  push fs:[00000024]              ;PEB->CurrentTID (!+)

  push esp                        ;lParam
  push fs:[00000020]              ;wParam: PEB->PID
  push WM_MONITOR_RETURN_HANDLER  ;wmMsg:  return handler
  push p_lSubClassFormHwnd        ;hWnd:   hwnd okna
  mov  eax, lpSendMessage
  call eax                        ;call SendMessage
  mov  dword ptr[esp], eax        ;eax == návratová adresa (1)

  push tCS.lpCriticalSection
  mov  eax, tCS.lpLeaveCriticalSection
  call eax
;-----------------------------------------
  add  esp, 4                     ;(2)(!-)
PopFD
PopAD                             ;(3)
pop dword ptr[esp-4]              ;odstranění ID háku z handleru

push dword ptr[esp-2C]            ;(4) uložená návr. adresa z (1)
ret
```

Jak je vidět, mnoho novinek zde není, avšak je zde použit nový trik. Problémem této funkce je, že musíme dostat nějakým způsobem „ven“ z chráněné části kódu hodnotu registru EAX **(1)** – což je návratová adresa, na kterou se bude pro pokračování programu skákat. Vzhledem k tomu, že všechny registry (včetně registru EAX) jsou následně přepsány zpět zálohou (popad), je jedinou možností, jak návratovou hodnotu propašovat ven (a neriskovat race-condition problémy, například použitím „globální“ proměnné), je uložení této hodnoty opět do zásobníku. Řádek **(1)** přepíše předtím vloženou hodnotu TID na návratovou adresu (jednoduše se tím vyhneme jednomu zbytečnému add a push). Následně je volán odchod z kritické sekce a nyní pozor – v této chvíli – řádek **(2)** ESP stále ukazuje na návratovou adresu v zásobníku. My však potřebujeme obnovit flagy a registry, které jsou pod návratovou adresou. Proto posuneme ukazatel ESP tak, aby ukazoval na zálohované flagy. Návratová adresa je tedy po provedení instrukce PopFD od ESP vzdálena -4 byty. Po provedení instrukce PopAD **(3)** ESP ukazuje na ID háku, který vložil do zásobníku handler (viz výše ReturnHook Handler (1)), ten je již potřeba odstranit šetrně k prostředí (tedy pomocí triku s pop-em). Po všech těchto roll-bacích je návratová adresa v zásobníku vzhledem k ESP o 11 DWORDů (4 * 11 = 44, tj. 0x2C ⇒ esp-2C) níže.

5.4.4. IAT / EAT hook handler

Jelikož je naším cílem vytvořit i monitor, můžeme přidat klasické hákování IAT i EAT, oba háky mají vzhledem ke své povaze stejnou implementaci.

```
CallHook Handler:
  push lHookId                                  ;(!+) id háku
  mov  dword ptr[esp-4], lpMonitorCallHandler
  call dword ptr[esp-4]                         ;call CallHandler
  pop  dword ptr[esp-4]                         ;(!-)
  mov  dword ptr[esp], lpReturnHookHandler      ;(1)
  push lpOriginalFunction                       ;skok na původní funkci
  ret

ReturnHook Handler:
  push lHookId                        ;(!+, odstraněno ReturnHandlerem)
  push lpMonitorReturnHandler         ;skok do ReturnHandleru
  ret
```

Jak je vidět, žádné novinky. Jelikož je skrze IAT vždy volána funkce pomocí instrukce CALL, pak je na vrcholu zásobníku (ESP) vždy návratová adresa. Ta je na řádku **(1)** přepsána na adresu ReturnHandleru.

5.4.5. Samotné monitorování – použití handlerů

Nyní stačí pouze jednotlivé handlery zapsat do paměti cílového procesu. Nejprve jsou zapsány CallHandler a ReturnHandler, následně jednotlivé handlery pro každý jednotlivý hák a jako poslední se provede samotné zaháknutí kódu / přepsání adresy v IAT/EAT.

Pokud vše proběhlo správně, stává se z cílového procesu proces monitorovaný, odesílající data pomocí funkce `SendMessage` Monitorovací aplikaci. Jediné, co tedy zbývá, je přijímač v Monitorovací aplikaci.

5.4.6. Přijímací handler Monitorovací aplikace

Přijímač Monitorovací aplikace je díky využití funkce `SendMessage` velice jednoduchou záležitostí, zvláště v jazycích typu C:

```
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
  switch(message)
  {
    case WM_MONITOR_CALL_HANDLER:
      printf("Volani monitorovane fce v procesu id = %d", wParam);
      ...
      system("PAUSE");
  return true;
    break;
    ...
  }
}
```

Tento jednoduchý kód Monitorovací aplikace způsobí, že při volání funkce hákované pomocí monitorovacího breakpointu bude monitorovaný program pozastaven, dokud uživatel v Monitorovací aplikaci nestiskne libovolnou klávesu.

5.5. Kompletování Monitorovací aplikace

Vytvořit kompletní Monitorovací aplikaci je nyní otázkou několika málo funkcí, z nichž jsme již řadu zmínili a probrali. V této kapitole projdeme schémata všech klíčových funkcí vedoucích k realizaci celé Monitorovací aplikace.

5.5.1. Inicializace práce

Nejprve je nutné práci s budoucím monitorovaným programem inicializovat. To obnáší několik málo kroků, z nichž je každý bezpodmínečně nezbytný pro krok následující. Jakákoliv chyba ve kterémkoliv kroku znemožňuje monitorování takového programu.

1. Je vybrán již běžící proces / specifikován proces k vytvoření, který je následně spuštěn (viz kapitola 5.3.4.) ⇒ cílový proces.
2. Je zkontrolováno, zda je cílový proces kompatibilní s podporovanou instrukční sadou Monitorovací aplikace.
3. Jsou načteny základní informace o cílovém procesu, jako aktuálně nahrané moduly a další užitečné informace.
4. Je ověřeno, zda má cílový proces nahranou knihovnu user32.dll. Pokud ne, je pomocí DLL injektáže (viz kapitola 5.3.6.) nahrána.
5. Je alokována paměť pro handlery a paměť pro uložení struktury kritické sekce.
6. Do alokované paměti je zapsán kód pro inicializaci kritické sekce a spuštěn pomocí vzdáleného vlákna.
7. Do alokované paměti jsou zapsány základní handlery: CallHandler a ReturnHandler.

5.5.2. Monitorování cílového procesu

Po úspěšné inicializaci již máme volnou ruku v monitorování prakticky čehokoliv dle samotných schopností Monitorovací aplikace.

Inline hákování

Pomocí inline hákování – přepisování kódu cílového procesu – jsme vyřešili *monitorovací* a *debuggerovské breakpointy*. Pro jejich zavedení platí následující schéma:

1. Uživatelem je jednoznačně specifikována funkce/adresa, na které má být hák (breakpoint) umístěn. Jméno funkce je přeloženo na adresu pomocí mezi-procesní funkce GetProcAddress (viz kapitola 5.3.5., str. 44).

2. Je ověřeno, zda je na této adrese stránka přepisovatelná. Pokud není, je upravena na přepisovatelnou.

3. Postupně jsou na této adrese čteny instrukce, které budou přepsány skokem, a je tedy nutné je zálohovat.

4. Je přidán (zapsán) nový handler (viz kapitola 5.4.2.) obsahující opravené instrukce a unikátní identifikátor háku.

5. Kód je na specifikované adrese přepsán na instrukci skoku, který vede na tento handler.

IAT a EAT hákování
IAT a EAT háky jsou si velice blízké, avšak je nutné dát si pozor na dvě věci. 1) adresy v IAT jsou absolutní (ukazují přímo na počátek funkce v paměti), zatímco adresy uvedené v EAT jsou relativní k adrese počátku modulu.
2) pokud chceme hákovat IAT, tedy tabulku importů v daném modulu, musíme uvést tři údaje. 1. modul, ve kterém chceme IAT hákovat, 2. jméno funkce a 3. jméno knihovny, která tuto funkci exportuje. To proto, že například soubor xyz.exe může importovat knihovny a.dll a b.dll, přičemž obě mohou exportovat stejně pojmenovanou funkci – proto je nutné je rozlišit. U EAT tento problém logicky nenastává. Zaháknutí IAT/EAT se provádí následovně:

1. Je načten IAT/EAT modulu, ve kterém chceme hák provést.

2. V IAT/EAT nalezneme záznam odpovídající specifikované funkci.

3. Je přidán handler obsahující skok na původní adresu (která bude v IAT/EAT přepsána) a unikátní identifikátor háku.

4. Záznam v IAT/EAT je přepsán na adresu handleru.

Nyní jsou všechny háky aktivní a přeposílají informace do specifikovaného okna Monitorovací aplikace skrze WM zprávy.

5.5.3. Přijímání zpráv Monitorovací aplikací

Přijímání zpráv od monitorovaného procesu se provádí stejně, jako správa jakýchkoliv jiných WM zpráv ve funkci `WindowProc`. Pokud programovací jazyk implicitně tuto funkci neuvádí, je možné využít funkci `SetWindowLong` s parametrem `GWL_WNDPROC`, která subclassuje (hákuje) původní funkci `WindowProc` do zadané nové funkce [5]. V technologiích .NET je možné využít klíčového slova `Overrides` k definování vlastní funkce `WindowProc`. Příklad přijímání zpráv Monitorovací aplikací je uveden v kapitole 5.4.6.

6. Výsledná Monitorovací aplikace

S využitím postupů probraných v kapitole 5. věnované různým technikám hákování jsme vytvořili aplikaci, která naplňuje cíle vytyčené v kapitole 4. Základním požadavkem pro nás byla schopnost pozastavit běh monitorovaného programu při volání nebo návratu uživatelem specifikované funkce a zároveň schopnost editovat její parametry, návratovou hodnotu, kontext vlákna a paměť. Této funkčnosti bylo dosaženo a dále byla rozšířena tak, že se schopnosti výsledné Monitorovací aplikace blíží klasickým debuggerům. Na rozdíl od klasických debuggerů však Monitorovací aplikace není, díky jinému principu fungování, detekovatelná žádným z běžných triků.

Monitorovací aplikaci lze použít jako klasický monitor importovaných, ale i lokálních funkcí, a to pomocí všech oblíbených metod, kterými jsou inline hákování a hákování tabulek IAT a EAT. Tato základní funkčnost je rozšířena i o důležitou schopnost pozastavit analyzovaný program při volání nebo návratu monitorované funkce, a tak dovolit uživateli měnit její vstupní i výstupní argumenty, případně i jakoukoliv jinou paměť. Po uživateli navíc tato funkčnost nevyžaduje prakticky žádné výjimečné programátorské znalosti. Konkrétní ukázku použití monitorování API funkcí naleznete v příloze A.5.1. Monitorovací aplikaci však lze použít i jako (téměř plnohodnotný) debugger na způsob OllyDbg[24]. Na jednotlivé řádky kódu programu lze nastavovat breakpointy, a tím sledovat a kontrolovat běh analyzovaného programu stejně jako v debuggeru, a to včetně možnosti jednoduše editovat hodnoty registrů a flagů procesoru, zobrazovat a editovat obsah zásobníku a libovolné další paměti. Monitorovací aplikace dále nabízí i další zajímavou funkčnost, kterou je role frameworku. Ten umožňuje snadno doprogramovat libovolnou hotovou aplikaci o cokoliv dalšího tím, že v uživatelem specifikovaném místě přesměruje „tok kódu“ z monitorovaného programu přímo do uživatelem vytvořené DLL knihovny, čímž mu umožní absolutní kontrolu nad tím, co se v aplikaci děje. Příkladem takového doprogramování může být vytvoření kontroly pravopisu pro aplikaci Poznámkový blok, nebo detektor min pro hru Hledání min. Nový doprogramovaný kód navíc není vložen do monitorovaného programu, a tedy není jednoduše detekovatelný. Detaily, jak takové knihovny vytvořit naleznete v příloze A.6.

Výsledná Monitorovací aplikace je tedy vhodným nástrojem k reverznímu inženýrství libovolného software – především virů, které ji nejsou schopny odhalit pomocí běžných postupů.

[24]Téměř plnohodnotný proto, že ne vždy lze použít funkcionalitu single-step – krokování programu po jedné instrukci. Plnohodnotná implementace této funkčnosti by vyžadovala další zkoumání a kalkulace vyžadující velmi složité úpravy handlerů.

Závěr

Pokud shrneme výsledky z finální implementace celého prezentovaného konceptu, pak se dá s jistotou mluvit o úspěchu. Vytyčený cíl – zaplnit mezeru mezi monitory a debuggery aplikací, která bude svými schopnostmi konvergovat ke schopnostem klasických debuggerů bez použití klasických debug API, byl rozhodně dosažen. Svým konceptem je výsledná Monitorovací aplikace se svými schopnostmi totožná s klasickými monitorovacími aplikacemi, a zároveň nabízí možnost chovat se stejně, nebo přinejmenším velice podobně jako debugger – ovšem s jistou šetrností. Například běžný debugger není v uživatelském režimu schopen krokovat a jakkoliv kontrolovat běh subsystému Windows CSRSS – pomocí nového konceptu tento úkol není žádný problém. Díky úplné absenci jakýchkoliv principů klasického debuggeru je Monitorovací aplikace téměř nedetekovatelná žádnými z běžných anti-debuggovacích triků – výjimku tvoří pouze triky založené na sledování času v kritických místech (ty mají za úkol odhalit, zda není kód zkoumán v pozastaveném stavu – aktivním breakpointu) a detekce založené na testování změn v kódu programu (nepříliš časté). Navíc je možné Monitorovací aplikaci používat současně s dalším debuggerem – což u běžných debuggerů není možné. Díky těmto vlastnostem je tak Monitorovací aplikace vhodná pro zkoumání virů a jiných nebezpečných programů, které chrání samy sebe před reverzním inženýrstvím a jejich analýza běžnými nástroji by byla příliš složitá či zdlouhavá, což bylo právě důvodem k vypracování této práce.

Monitorovací aplikace vzhledem ke svému konceptu dovoluje i přemostit „tok kódu“ monitorovaného programu do zadané DLL knihovny načtené v Monitorovací aplikaci (plugin) – jednoduše řečeno – pomocí Monitorovací aplikace se dá doprogramovat libovolný již hotový program o libovolný kód. Tato schopnost je při zkoumání virů a jiného škodlivého software velkou výhodnou, avšak její využití plně závisí na kreativním uchopení daného problému analytikem.

Výsledná Monitorovací aplikace má však mnoho bodů, které mohou být vylepšeny nebo dodělány. Především se pak jedná o plnou realizaci krokování po jednotlivých instrukcích (velice zesložiťující handler). Dále se například jedná o přidání ukládání stavu Monitorovací aplikace a monitorovaného programu, o lepší podporu monitorování více procesů zároveň, vytváření grafů znázorňujících jejich vazby, historii, jednoduché skripty, načítání debug informací vytvořených kompilátorem, propojení s IDA. S dalšími úpravami by bylo možné používat Monitorovací aplikaci i k testování a penetrování funkcí programů.

Conclusions

If we summarize the results of the final implementation of the presented concept, then we can speak with certainty about success. The goal – fill the gap between the monitors and debuggers by the new Application with abilities close to conventional debuggers without usage of conventional debug API was definitely achieved. This concept offers the same functionality as monitoring applications and offers (nearly) the same abilities as common debuggers, but with some care. For example, a common debugger is not able (in user mode) to debug running Windows subsystem – CSRSS. This task is not problem for tis new concept. Thanks to the complete absence of any principles of classical debugger, Monitoring application is almost undetectable by any of the conventional anti-debugging tricks. It is also possible to use Monitoring application together with another debugger – what is with conventional debuggers not possible. These features are thus suitable for application monitoring, study of viruses or other malicious programs that protect themselves against reverse engineering.

Monitoring application due to its concept allows bridging the flow of code from monitored program code to the specified DLL loaded in Monitoring application (plugin) – simply put – it is possible to append any code to existing program. This ability is very interstring for analysation of viruses and other malicious software, but its use depends entirely on creativity of the analyst.

The resulting Monitoring application, however, has many points that can be improved or resolved. First of all it is about the full realization of single-step breakpoints. Further, for example, better support of monitoring multiple processes simultaneously, creating charts showing their links, history, simple scripts, loading debug information generated by the compiler, link with the IDA. With additional modifications would be possible to use Monitoring application as well as function testing and probing programs.

Reference

[1] *Merriam-Webster Dictionary*
http://www.merriam-webster.com/dictionary/reverse%20engineer

[2] *Microsoft Developer Netword (MSDN)*
http://msdn.microsoft.com

[3] *Iczelion's Win32 Assembly Homepage*
http://win32assembly.online.fr

[4] *MASM 32 – Intel Opcodes And Mnemonics*
http://masm32.com/

[5] *Microsoft Developer Netword (MSDN)*
http://msdn.microsoft.com

[6] Zemánek, Jakub. *Cracking bez tajemství.* Computer Press, Praha, 2002.

A. Uživatelská dokumentace

A.1. Instalace

A.1.1. Požadavky pro běh aplikace

- Windows 2000, Windows XP, Windows Vista, Windows 7 **32** a 64 bit[25]
- VB6 Run-Time (implicitně instalován ve všech verzích Windows)
- LDebugger.exe a přibalené soubory MSCOMCTL.OCX, RICHTX32.OCX, riched20.dll, riched32.dll
- *all_apis.txt* (volitelné)

A.1.2. Instalace

Aplikace je plně portable – tzn. nevyžaduje instalaci. Jediný vyžadovaný zásah (pokud je nutný) je registrace použitých OCX knihoven (RICHTX32.OCX, MSCOMCTL.OCX), která je provedena při spuštění aplikace.

Pokud v systému není přítomen VB6 Run-Time, je možné jej stáhnout a instalovat z: http://www.microsoft.com/download/en/details.aspx?id=24417. Instalační soubor VB6 Run-Time Redistributable Pack (VB6.0-KB290887-X86.exe) je přiložen i na přiloženém CD ve složce INSTALL.

[25] Monitorovací aplikaci lze spustit na 64bitových systémech, ale musí být použita pro monitorování čistě 32bitových aplikací.

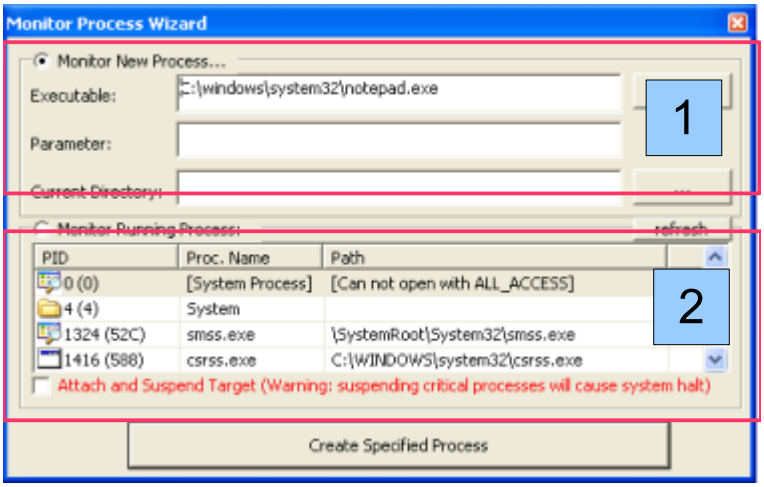
A.2. Spuštěná aplikace

Program se spouští souborem LDebugger.exe. V případě, že nejsou registrovány důležité OCX knihovny, je nutné spustit LDebugger.exe jako Administrátor (pravý klik na LDebugger.exe → Spustit jako Administrátor) – na tento fakt a další kroky upozorní sama aplikace.

Aplikace startuje do okna Monitor Process Wizard (File → New Monitor).

A.3. Monitor Process Wizard

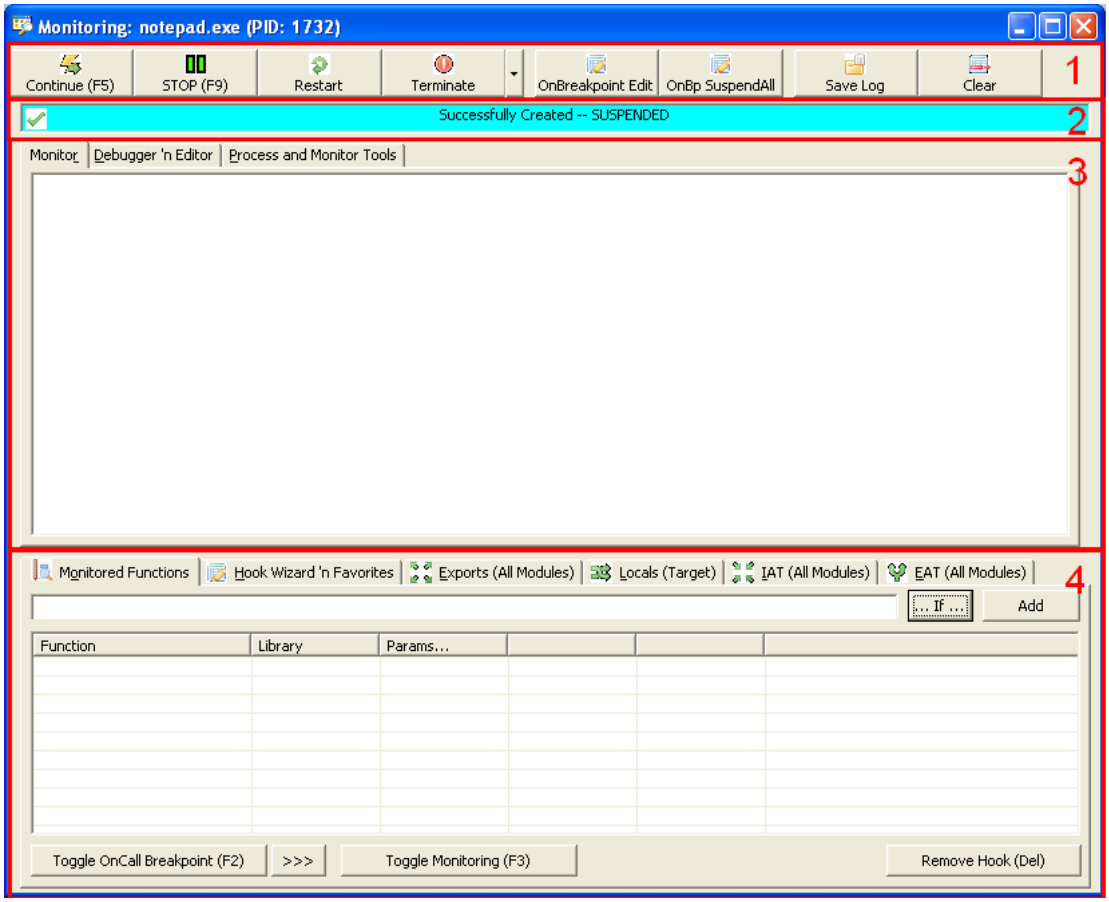
Zde uživatel specifikuje a) **proces, který má být spuštěn** (1), jeho parametry a konstantu Current Directory (v jakém umístění je program spouštěn), b) vybere již **existující proces**, ke kterému se chce připnout (Attach) (2) – pokud uživatel nemá k některému z procesů přístupová práva, je tento fakt uveden v políčku `Path` informací „`Can not open with ALL_ACCESS`“.
Upozornění: při připínání k procesu je možné zvolit, zda mají být při připnutí a breakpointech suspendovány všechna vlákna – tzv. mód pozastavování jednotlivých vláken.



Obrázek 11. Monitor Process Wizard – vytváření / připnutí k procesu.

A.4. Hlavní okno monitoru

Po vytvoření, nebo připnutí k procesu je zobrazeno Hlavní okno monitoru (pro každý monitorovaný program je vytvořeno vlastní okno, proto použijme název Monitorovací instance), to se skládá z Toolbaru (1), Statusbaru (2), Hlavní sekce (3) a Správy monitorovaných funkcí (4).



Obrázek 12. Hlavní okno – Monitorovací instance

A.4.1. Toolbar (1)

Toolbar obsahuje základní funkce pro správu Monitorovací instance a monitorovaného programu:

- Continue (F5) – uvolní k pokračování proces / jednotlivé pozastavené vlákno.
- STOP (F9) – pozastaví všechna vlákna procesu. Pokud byl při připínání k procesu ponechán mód pozastavování jednotlivých vláken, bude uživatel při kliknutí na toto tlačítko dotázán.
- Restart – *zatím nebylo implementováno*
- Terminate – ukončí monitorovaný proces s ExitCode 0. Pokud je potřeba specifikovat vlastní návratový kód, je možné použít drop-down menu a tlačítko **Terminate with specified ExitCode**.
- OnBreakpointEdit – pokud je zamáčknuto, při vyvolání breakpointu je nastavena jako aktivní záložka Hlavní sekce **Debugger**.
- OnBP SuspendAll – pokud zamáčknuto, při vyvolání breakpointu jsou pozastavena **všechna** vlákna.
- Save Log – uloží obsah Hlavní sekce **Monitor**.
- Clear – smaže obsah Hlavní sekce **Monitor**.

A.4.2. Statusbar (2)

Status bar zobrazuje aktuální stav Monitorovací aplikace a Monitorovaného programu.

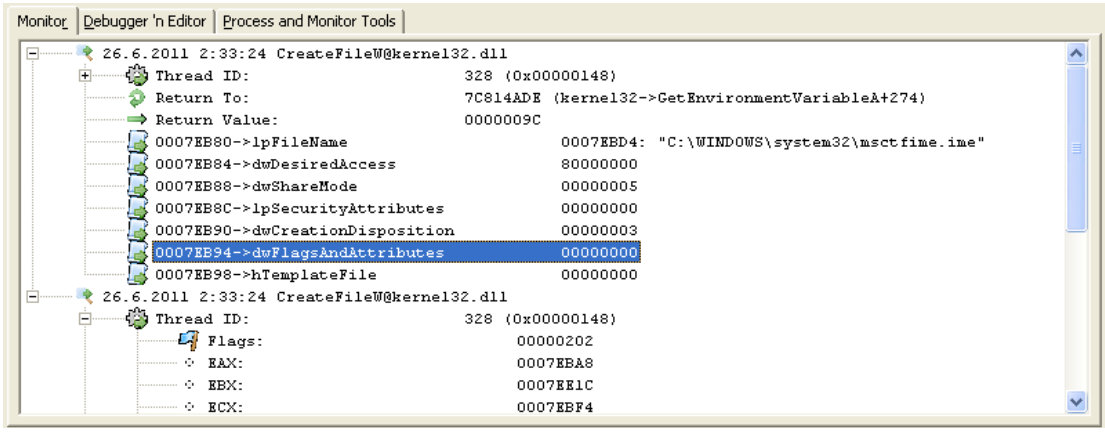
A.4.3. Hlavní sekce (3)

Hlavní sekce se skládá ze tří záložek: Monitor, Debugger a Process And Monitor Tools.

Monitor

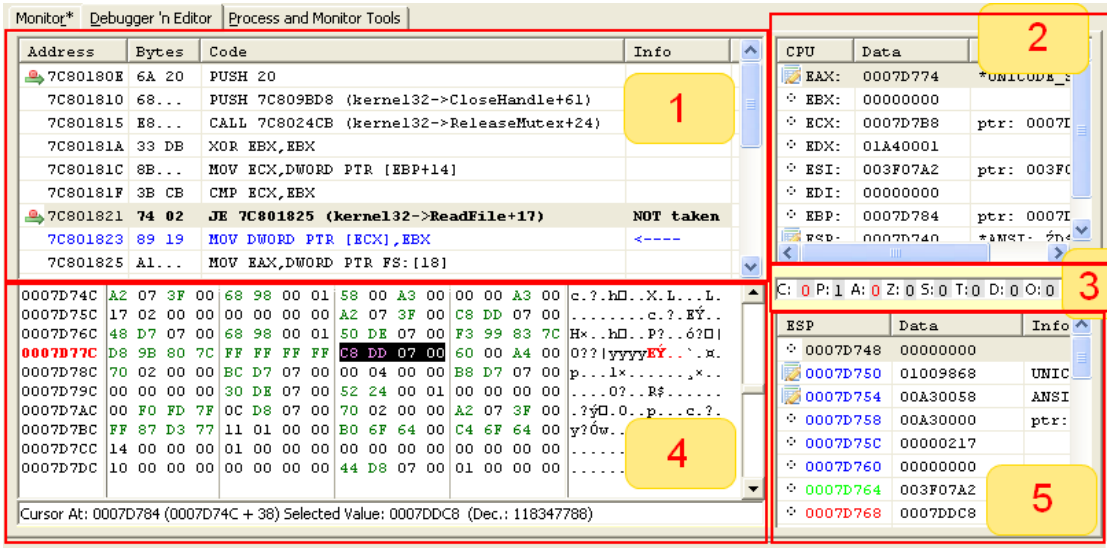
Zde jsou zobrazeny všechny výsledky monitorování (události – eventy monitoru) v chronologickém sledu. Zobrazeny jsou následující detaily: `Thread ID` – ID vlákna, které breakpoint spustilo a jeho kontext; `Return To` – adresa návratu (pouze u monitorování funkcí); `Return Value` – návratová hodnota funkce (pouze u monitorování funkcí, tato hodnota je vyplněna až při opouštění funkce); parametry – dle specifikace v deklaraci (pouze u monitorování funkcí – při návratu z monitorované funkce jsou parametry obsahující pointery znovu načteny).

Pokud event monitoru spustil breakpoint (a tedy pozastavil proces), je tento event označen modře.



Obrázek 13. Hlavní sekce – Monitor

Debugger



Obrázek 14. Hlavní sekce – Debugger

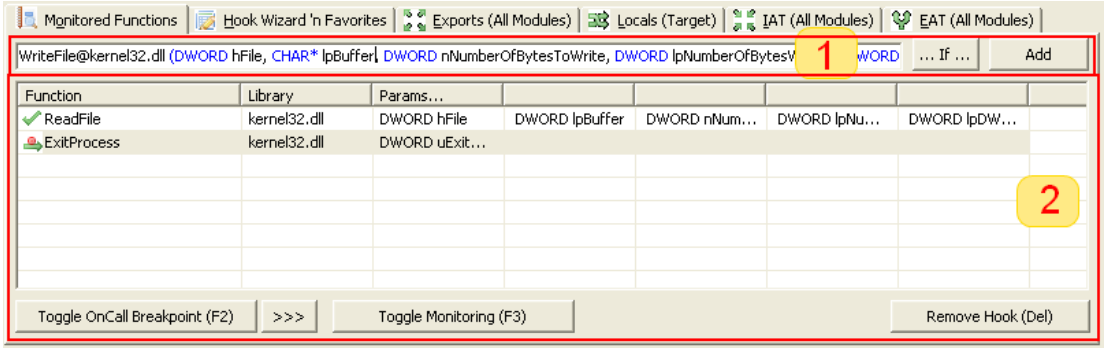
Záložka debugger nabízí všechny základní vymoženosti klasických debuggerů: disassembler (1), registry procesoru (2), flagy procesoru (3), editor paměti (memory editor) (4), zobrazení zásobníku (5).

Hodnoty jednotlivých registrů / hodnoty v zásobníku / konstanty v disassembleru jsou na základě heuristiky testovány, zda nesou: pointery na ANSI / UNICODE string, pointer na strukturu `UNICODE_STRING`, pointer na modul nebo funkci, nebo pouze validním pointerem.

- Disassembler: kliknutí na řádek s podmíněným skokem – za běhu: zobrazení, kam bude skok skákat; při breakpointu – zobrazení, kam bude skákat na základě vyhodnocení podmínky. Stisknutím klávesy F2 je na řádek nastaven breakpoint – pozor: breakpointy vložené debuggerem a monitorem mezi sebou nelze převádět (pokud je na řádku nastaven breakpoint monitoru, pak nemůže být přepsán na debuggerovský breakpoint).
- Registry procesoru: dvojklik – editace hodnoty registru.
- Flagy procesoru: klik – změnění hodnotu flagu.
- Editor paměti: editovat paměť je možné dvojklikem na byte, od kterého se má paměť přepisovat. Editovat konkrétní hodnotu je možné jejím označením a následným dvojklikem.
- Zobrazení zásobníku: zobrazení jednotlivých hodnot zásobníku. Editovat hodnotu je možné dvojklikem. Zobrazení hodnoty v editoru paměti je možné kliknutím na hodnotu zásobníku se stisklou klávesou SHIFT. Zobrazení obsahu pointeru je možné dvojklikem na hodnotu zásobníku se stisknutou klávesou SHIFT.

Process and Monitor Tools
Karta pro další nástroje spojené s monitorovaným procesem a správou Monitorovací aplikace. Obsahuje možnost injektáže libovolné DLL knihovny do monitorovaného procesu a načtení pluginu do Monitorovací aplikace. Detaily k vytváření pluginů pro Monitorovací aplikaci můžete nalézt v samostatné části Pluginy A.6.

A.4.4. Správa monitorovaných funkcí (4)



Obrázek 15. Správa monitorovaných funkcí

Monitored functions
Zde je možné přidávat funkce určené k monitorování. Přidání funkce se provádí vložením deklarace do textového pole (1). Pomocí tlačítka `[...IF...]` je možné přidat k deklaraci rozšiřující podmínky. Funkce je přidána po stisknutí tlačítka `[Add]`, nebo klávesy ENTER. Všechny monitorované funkce jsou uvedeny v seznamu (2) – zde je možné nastavit jednotlivým funkcím různé druhy breakpointů pomocí tlačítka `[>>>]`, vypnout/zapnout vypisování eventů pomocí tlačítka `[Toggle Monitoring]` (funkce bude i nadále monitorována, pouze nebudou zobrazovány záznamy) a odstranit funkci z monitorování.

Úprava deklarace již přidané funkce je možná pomocí dvojkliku na záznam v seznamu (2), deklarace je přenesena do textového pole (1) a zde je možné ji upravit, modifikace je potvrzena tlačítkem `[Add]`.

Deklarace funkce má následující syntax:

```
[TYP] Funkce@Knihovna (#registr#TYP param1, ...) $podmínky$
```

- **TYP** proměnné a návratové hodnoty:

Typ	**Označení**	**Ukazatel**
4 byte	**handle**, **dword**, int, int32, integer, long	**dword***, int*, integer*, long*, dwordptr
2 byte	**word**, int16, short	**word***, int16*, short*
1 byte	byte, **char**	**byte***, byteptr
ANSI String	n/a	**char***, strptr, ansi, lpcstr
UNICODE String	n/a	**wchar***, wstrptr, unicode lpcwstr
UNICODE_STRING	n/a	**unicode_string**, bstr, *bstr

- **Funkce@Knihovna**: jméno exportované Funkce z Knihovny. (Dvě různé knihovny mohou exportovat stejně pojmenované funkce, proto je nutné specifikovat hosta.) Pokud není funkce exportována nebo je nutné zaháknout libovolnou adresu, je použit tvar: `adresa@address`.

```
Beep@kernel32.dll ...
0047BC14@address ...
```

- **Parametry** jsou specifikovány dle jejich umístění v registru / zásobníku. Neuvedení registru → zásobník.
  stdcall – všechny parametry v zásobníku:

  ```
  ... (HANDLE hFile, char* lpBuffer, ...) ...
  ```

  Microsoft FastCall – první parametr ECX, druhý EDX, zbytek v zásobníku:

  ```
  ... (#ECX#char* destination, #EDX#char* source, DWORD bufferlen) ...
  ```

- **Podmínky** specifikují, kdy má být provedena specifikovaná akce:

  ```
  $AKCE: podmínka1, podmínka2, ...$
  ```

  Akce:

  - OnlyIf – event je zobrazen pouze tehdy, když je splněna podmínka
  - BreakIf – breakpoint je aktivován pouze při splnění podmínky
  - IgnoreIf – pokud je splněna podmínka, nebude event zobrazen

  ```
  $OnlyIf:  velikost==123, soubor==test.txt$
  $BreakIf: soubor==jiny.jpg$
  ```

  Příklady:

- ```
  Beep@kernel32.dll (DWORD dwFreq, DWORD dwDuration) _
            $BreakIf: dwDuration == 150$
  ```

  Monitorovaný proces bude pozastaven, pokud bude volat funkci Beep z knihovny kernel32 s libovolným parametrem `dwFreq` a `dwDuration == 150`.

- ```
  wchar* GetEnvironmentStringsW@kernel32 ()
  ```

  Monitorovací aplikace zobrazí textový řetězec vrácený funkcí GetEnvironmentStrings.

Hook Wizard 'n Favorites
Zde jsou uvedeny všechny entrypointy modulů a tématicky řazené důležité funkce – ty jsou všechny jednoduše připraveny na zaháknutí. Stačí pouze zaškrtnout skupinu / jednotlivé funkce, nebo entrypointy a kliknout na tlačítko `[Hook selected functions=>]`. Kliknutím na funkci a stisknutím klávesy ENTER přenesete deklaraci funkce k přidání do karty Monitored Functions. Doporučená technika háku: přepsání kódu (inline hák).

Exports (All Modules)
Zde jsou uvedeny exporty všech načtených modulů (seznam je aktualizován po každém kliknutí na tuto kartu). Jednotlivé exporty nemají vlastní předdefinované deklarace parametrů – ty musí doplnit uživatel. Kliknutím na exportovanou funkci a na tlačítko `[Prepare Inline Hook]` (nebo stisknutím klávesy ENTER) přenesete kostru deklarace k přidání do karty Monitored Functions. Doporučená technika háku: přepsání kódu (inline hák).

Locals (Target)
Zde jsou uvedeny lokální funkce cílového monitorovaného programu, které byly rozpoznány základní heuristikou. Kliknutím na lokální funkci a na tlačítko `[Prepare Inline Hook]` (nebo stisknutím klávesy ENTER) přenesete kostru deklarace k přidání do karty Monitored Functions. Doporučená technika háku: přepsání kódu (inline hák).

IAT (All Modules)
Zde jsou uvedeny všechny tabulky importů (IAT) všech modulů (seznam je aktualizován po každém kliknutí na tuto kartu). Kliknutím na importovanou funkci a na tlačítko `[Prepare Address Hook]` (nebo stisknutím klávesy ENTER) přenesete kostru deklarace k přidání do karty Monitored Functions. Technika háku: IAT hákování.
Poznámka k deklaraci: deklarace funkce pro IAT hákování je obohacena o prefix

```
[IAT@knihovna s cílovým IAT] funkce@knihovna.dll ()
```

protože knihovna, ve které chceme hákovat IAT, není ta samá, která exportuje funkce (tj. knihovna uvedená za @).

IAT (All Modules)
Zde jsou uvedeny všechny tabulky exportů (EAT) všech modulů (seznam je aktualizován po každém kliknutí na tuto kartu). Kliknutím na exportovanou funkci a na tlačítko `[Prepare Address Hook]` (nebo stisknutím klávesy ENTER) přenesete kostru deklarace k přidání do karty Monitored Functions. Technika háku: EAT hákování.

A.4.5. Ukončování monitorování

Upozornění: vzhledem ke konceptu fungování Monitorovací aplikace, je nutné, aby byly provedeny veškeré návraty z monitorovaných funkcí před ukončením Monitorovací aplikace! Jinak hrozí (skrze suplování zásobníku s návratovými adresami) pád monitorovaného programu. Při ukončování instance Monitorovací aplikace jsou odstraněny z paměti monitorovaného programu všechny háky.

A.5. Příklady práce

V této kapitole jsou uvedeny příklady práce s Monitorovací aplikací.

A.5.1. Základy – Poznámkový blok

Cílem bude přepsat zapisovaný text Poznámkovým blokem.

1. Spustit Monitorovací aplikaci,

2. spustit Poznámkový blok a v Monitorovací aplikaci, v okně Monitor process Wizard klepnout na tlačítko `[Refresh]`, v seznamu běžících procesů označit notepad.exe a kliknout na tlačítko `[Attach To Selected Process]`.
   TIP: procesy jsou v seznamu řazeny chronologicky dle jejich spuštění.

3. Vyčkat, dokud nejsou načteny všechny detaily v monitorovací instanci. Po dokončení inicializace je statusbar zelený s nápisem „Process is running...".

4. Nyní stačí do textového pole pro deklarace vložit deklaraci funkce, kterou chceme kontrolovat – zápis do souboru, tj. `WriteFile` z knihovny kernel32.dll:

   TIP: pro vložení deklarace z nápovědy je možné použít kurzorovou klávesu →.

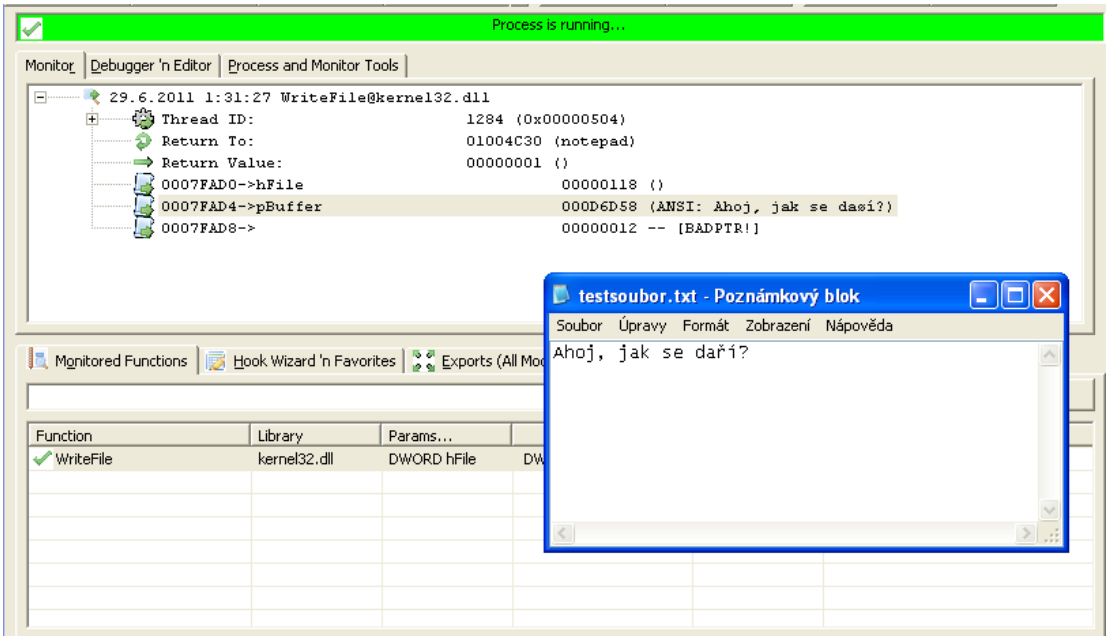
   ```
   WriteFile@kernel32.dll (DWORD hFile,
                           DWORD pBuffer,
                           DWORD nNumberOfBytesToWrite, ...)
   ```

   Víme, že parametr `pBuffer` v Poznámkovém bloku ukazuje na čistý text, avšak nevíme, zda jde o ANSI, nebo UNICODE. Typ parametru `pBuffer` můžeme ponechat jako DWORD, Monitorovací aplikace se sama pokusí typ při volání odhadnout. Deklaraci vložíme tlačítkem `[Add]`.

5. Nyní vyzkoušíme, zda byla zaháknuta správná funkce: do Poznámkového bloku napíšeme libovolný text (nejlépe však větu obsahující několik slov, aby bylo pro heuristiku Monitorovací aplikace snadnější určit, zda jde o ANSI, nebo o UNICODE) a soubor uložíme.

6. Do výpisu monitorovaných událostí přibylo volání funkce `WriteFile`, kliknutím na [+] zobrazíme detaily tohoto volání, měl by se naskytnout obdobný pohled (Obrázek 16.):

7. Jelikož chceme změnit text, který bude zapisován, musíme nastavit na volání této funkce breakpoint. To provedeme kliknutím na funkci `WriteFile` uvedenou v seznamu Monitored functions 15. a stisknutím klávesy F2, nebo tlačítka `[Toggle OnCall Breakpoint]`. Ikonka funkce se změní na šipku s červenou tečkou. Nyní bude běh Poznámkového bloku pozastaven vždy na začátku funkce `WriteFile`.

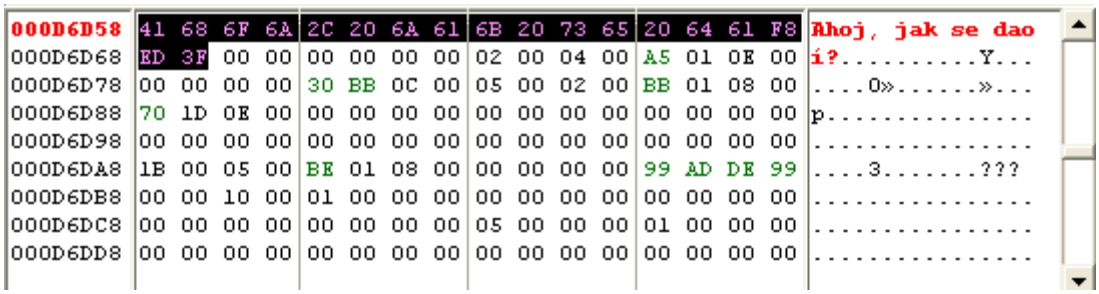


Obrázek 16. Zachycené volání API funkce WriteFile

8. Znovu nechme v Poznámkovém bloku uložit obsah.

9. Poznámkový blok je pozastaven, statusbar monitorovací instance je modrý a informuje o aktivování breakpointu.

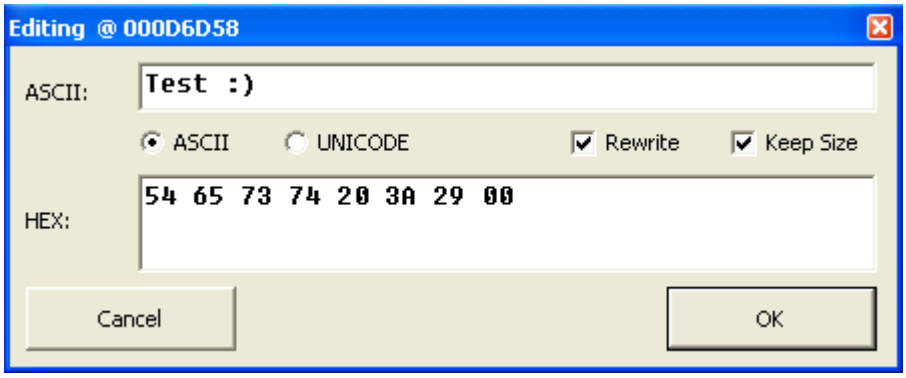
10. Protože chceme editovat text uložený na adrese specifikované `pBuffer`, přesuneme se do karty Debugger 'n Editor. Zobrazení obsahu parametru `pBuffer` v editoru paměti je možné dvěma způsoby: 1) kliknutím do editoru paměti, stisknutí kláves CTRL + G, jako adresa je použita hodnota parametru `pBuffer` (druhá položka v zásobníku), nebo 2) dvoklikem se stisknutou klávesou SHIFT na položku `pBuffer` v zásobníku. Nyní stačí tažením kurzoru označit v hex dumpu celý ukládaný text (Obrázek 17.):



Obrázek 17. Editace textu v memory editoru

Dvojitým kliknutím na označený text je vyvoláno editovací okno. Editovací

okno předvoleně hlídá délku vstupu, aby nedošlo k přepsání uživatelem neoznačené paměti. Text může být přepsán na cokoliv jiného, osobně zvolím řetězec „Test :)“ (Obrázek 18.):



Obrázek 18. Editace textu v memory editoru

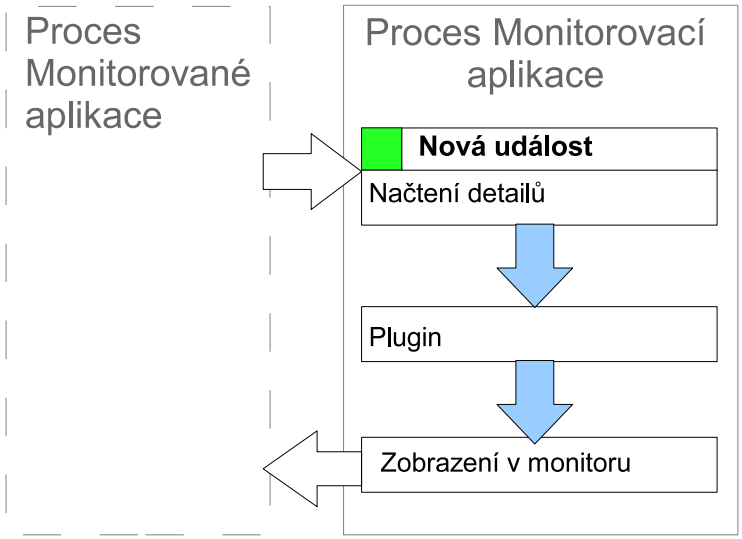
Pro dokončení editace a zápis do paměti stačí kliknout na tlačítko `[OK]`. Poznámka: pokud je nově přepsaný text kratší než původní, je nutné upravit i parametr `nNumberOfBytesToWrite` specifikující délku zapisovaných dat. Editovat tento parametr je možné dvojklikem na jeho položku v zásobníku. Pro řetězec „Test :)“ je nová délka 7 bytů.

11. Nyní stačí Poznámkový blok opět spustit (uvolnit) pomocí klávesy F5, nebo tlačítka v toolbaru `[Continue]`.

12. Otevřte uložený soubor. Obsah byl při zapisování změněn.

A.6. Pluginy

Monitorovací aplikace dovoluje používat i vlastní DLL pluginy. Pluginu jsou předávány veškeré události získané Monitorovací aplikací, včetně všech nezbytných informací k jejich úpravě. Komunikaci ilustruje Obrázek 19.:



Obrázek 19. Plugin funguje jako filtr.

Návratová hodnota pluginu pak specifikuje, zda jsou informace zapsány do monitorovaných událostí, nebo zda budou vyfiltrovány.

A.6.1. Vytvoření pluginu

Monitorovací aplikace komunikuje s pluginem pomocí dvou STDCALL exportovaných funkcí:

```
DWORD GetPluginVersion(DWORD res1, DWORD res2, DWORD res3, DWORD res4)
```

Funkce `GetPluginVersion` slouží pouze k předání verze pluginu Monitorovací aplikaci – tato funkce je volána při načtení pluginu do Monitorovací aplikace. Všechny parametry jsou rezervované s hodnotou NULL.

```
DWORD NewMonitorEvent(DWORD hProcess, DWORD PID,
                      DWORD ThreadID, EVENT_IN *in)
```

Funkce `NewMonitorEvent` je pak volána při každé události (volání a návrat). Parametr `hProcess` je handle monitorovaného procesu s právy `ALL_ACCESS` a právy měnit kontext vláken. PID specifikuje identifikátor monitorovaného procesu, ThreadID je identifikátor vlákna, které událost vytvořilo. Parametr `in` je ukazatel na strukturu nesoucí další informace:

```
struct EVENT_IN{
DWORD   eType;
LPCWSTR lpFunctionName;
LPCWSTR lpLibraryName;
CONTEXT *pContext;
  DWORD   lpReturnAddress;
  DWORD   dwReturnValue;
  DWORD   lpProcess_Parameters;
  DWORD   lpProcess_Context;
  DWORD   lpProcess_ReturnValue;
  DWORD   lpProcess_ReturnAddress;
unsigned int iParamCount;
LPWSTR   *params;
};
```

- `eType`: specifikuje typ události; 0 – volání, 1 – návrat, 2 – inicializace pluginu.
- `lpFunctionName`: pointer na jméno funkce (definováno dle uživatelské deklarace v Mon. apl.),
- `lpLibraryName`: pointer na jméno knihovny (definováno dle uživatelské deklarace v Mon. apl.),
- `pContext`: pointer na strukturu CONTEXT, **pouze ke čtení**.
- `lpReturnAddress`: návratová adresa funkce – pouze při volání,
- `dwReturnValue`: návratová hodnota funkce – pouze při návratu,
- `lpProcess_Parameters`: pointer na první parametr v monitorovaném procesu,
- `lpProcess_Context`: pointer na DWORD prvního z registrů uložených pomocí instrukce PUSHAD (POZOR! Nejedná se o ekvivalentní strukturu s CONTEXT) v monitorovaném procesu,
- `lpProcess_ReturnValue`: pointer na DWORD specifikující návratovou hodnotu funkce,
- `lpProcess_ReturnAddress`: návratová adresa volání, pouze při volání, pouze ke čtení,
- `iParamCount`: počet parametrů,
- `params`: pointer na pole obsahující pointery na **zpracované** řetězce parametrů.

Návratová hodnota `NewMonitorEvent`: 0 – nedojde k přidání události do výpisu (odfiltrování), 1 – přidá do výpisu.

Použití pluginů je prakticky všestranné – je možné je použít jako filtr s rozsáhlými podmínkami (na které ty vestavěné v Monitorovací aplikaci nestačí) nebo přímo k doprogramování funkcí monitorovaného programu. K práci jsou přiložené dva ukázkové zdrojové kódy jednoduchých pluginů.

B. Programátorská dokumentace

Programovací jazyk byl vybírán z následující množiny: Assembler, C/C++ (nativní, .NET), C# (.NET), Delphi, Visual Basic (6, .NET), dle následujících kritérií:

1. jednoduché vytváření veškerého GUI a jeho interakce – neztrácet čas „programováním textboxů“ a věnovat se přímo problematikou (nehledě na treeview, které v čistém kódu mají průměrně kolem 30 000 řádků);
   zbývá: C/C++ .NET, C# .NET, Delphi, Visual Basic (6, .NET),

2. snadná a nedozorovaná (nemarshalovaná) práce s pamětí – kopírování struktur a polí z paměti cizího procesu do Monitorovací aplikace; zbývá: C/C++ .NET, Delphi, Visual Basic 6,

3. nativní kód – skrze možné využití na „čistých“ snapshotech virtuálních PC bez .NET a jiných frameworků s nutností instalovat service packy atd.;
   zbývá: Delphi, Visual Basic 6

Vzhledem k několikaletým zkušenostem s programovacím jazykem Visual Basic 6 byl vybrán právě ten, a to i po zvážení všech nevýhod plynoucích z tohoto rozhodnutí. Pro jeho kompilaci je nutné použít Visual Studio 6. Bylo použito objektové paradigma.

B.1. Formuláře

Třídy vytvářející grafické prostředí:

`frmMain:`	hlavní rodičovské MDI okno.
`frmStartWizard:`	okno Monitor Process Wizard.
`frmMemoryEditor_Edit:`	okno Edit pro zjednodušení editace řetězců a binárních dat v editoru paměti.
`frmMonitorEventIf_Edit:`	GUI editor podmínek k deklaracím funkce.
`frmSubclassForm:`	subclassované okno přijímající WM zprávy o událostech v monitorovaném programu.

B.1.1. frmMonitorInstance

Monitorovací instance + GUI, vytváří prostředníka mezi uživatelem a rozhraním pro práci s monitorovaným programem (`clsMonitor`).

- `StartNewMonitor(...)`
  Předává rozhraní informace k vytvoření nového procesu.
- `StartAttachMonitor(...)`
  Předává rozhraní informace k připnutí k existujícímu procesu.
- `HandleCall(...)`
  Event zpracovávající událost volání.
- `HandleReturn(...)`
  Event zpracovávající událost návratu.

B.2. Třídy

B.2.1. clsAssembler32

Třída obsahující zjednodušený překladač 32-bitových assemblerovských instrukcí a s nimi spjatých dalších pomocných funkcí:

- `Assembly(...) As String`
  Přeloží zadanou instrukci na strojový kód relativně k zadané adrese.
- `IsConditionedJumpSatisfied (instrukce, flagy procesoru)`
  Vrací, zda je zadaný podmíněný skok splněn.
- `FixRelativeJump(...) As String`
  Opravuje zadanou relativní instrukci na absolutní.
- `AnalyzeCode (...)`
  Analyzuje kód zadaného modulu na obsah lokálních privátních funkcí.

B.2.2. clsDisassembler

Třída pro překlad 32-bitového strojového kódu do mnemonické podoby (disassembler): open source disassembler, autor Vanja Fuckar, email: inga@vip.hr.

`DisAssemble (...)` – překládá strojový kód na mnemonickou podobu.

B.2.3. clsMonitor

Nejdůležitější třída zaštiťující veškerou funkčnost Monitorovací aplikace.

Proces:

- `LoadSuspendedProcess (...)` – načte proces vytvářený pomocí API `CreateProcess` s flagem `CREATE_SUSPENDED`.
- `SuspendProcess (...)` – pozastaví proces (všechny / zadané vlákno).
- `ResumeProcess (...)` – uvolní proces (všechny / zadané vlákno).
- `MemEdit_RecognizeContent (...)` – funkce pro rozpoznávání obsahu paměti.
- `MemEdit_... (...)` – funkce pro základní práci s pamětí monitorovaného procesu.
- `ResolveFunctionToAddress (...)` – překlad jména funkce na její adresu v kontextu monitorovaného procesu.

Monitoring:

- `MonitorNewProcess (...)` – vytvoří, načte a inicializuje práci s novým procesem.
- `MonitorRunningProcess (...)` – inicializuje práci s již existujícím procesem.
- `HookInlineFunction (...)` – vytvoří inline hák (přepsání kódu) na zadanou adresu.
- `HookIATEATFunction (...)` – vytvoří IAT / EAT hák na zadanou funkci.
- `UnHookFunction (...)` – odstraní zadaný hák a obnoví původní hodnoty.
- `Handler_ProcessCall (...)` – funkce zpracovávající přijatou zprávu o volání.
- `Handler_ProcessReturn (...)` – funkce zpracovávající přijatou zprávu o návratu.
- `MakeMonitorCallHandler (...)` – vytvoří kód call handleru.
- `MakeMonitorReturnHandler (...)` – vytvoří kód return handleru.
- `MakeInlineHookHandler (...)` – vytvoří kód handleru inline háků (záloha přepsaných instrukcí, jejich fixace, ...).
- `MakeIATEATHookHandler (...)` – vytvoří kód handleru pro IAT / EAT háky.
- `RaiseNewMonitoredEvent (...)` – funkce pro vyvolání události `NewMonitoredEvent` informující o zachycení nové události. Informace o této události jsou předány pomocí parametrů.
- `HandleCallReturn (...)` – dle typu háku supluje vkládání do zásobníku návratových adres.
- `RequestCallReturnAddress (...)` – dle typu háku získává ze suplujícího zásobníku návratovou adresu.

Enumerátory:

- `GetAllExports (...)` – získá všechny exporty všech načtených modulů monitorovaného programu.
- `GetAllImports (...)` – získá všechny importy všech načtených modulů monitorovaného programu.
- `GetAllLocals (...)` – pokusí se pomocí heuristiky získat lokální funkce spustitelného souboru monitorovaného programu.
- `GetAllEntryPoints (...)` – získá všechny adresy entrypointů (adresy vstupů) všech načtených modulů monitorovaného programu.

Ostatní:

- `ProceedToPlugin (...)` – zašle data pluginu.
- `ParseDeclare (...)` – zpracuje deklaraci, výsledkem je struktura popisující deklarovanou funkci.
- `ProcessParamType (...)` – načte a upraví parametr ze zachycené funkce dle její deklarace. (Např. pokud je parametr pointer na string, tak tento string načte z paměti monitorovaného procesu).

B.2.4. clsResizer

Třída pro úpravu velikostí prvků ve formulářích.

- `AddControl (...)` – přidá prvek do seznamu prvků k přizpůsobování velikosti.
- `Resize() (...)` – funkce volaná při změně velikosti formuláře; přizpůsobuje prvky nové velikosti jejich rodičovského okna.

B.2.5. AutoCompleteIntelly

Automaticky doplňující textové pole.

- `FillListBoxByFile (...)` – naplní slovník k automatickému doplňování obsahem zadaného souboru.
- `SetAlternateDataSource (...)` – nastaví alternativní zdroj typu TreeView – pokud není nalezen zadávaný řetězec ve slovníku, je prohledán tento alternativní zdroj.
- `SetKeywords (...)` – nastaví klíčová slova, která budou při zadávání zvýrazněna modře.
- `EnterDown ()` – událost informující o stisknutí klávesy Enter.

B.2.6. DebuggerEditor

Uživatelský prvek (usercontrol) obsahující okna a zaštiťující práci s disassemblerem, editorem paměti, zásobníkem, registry a flagy.

- `InitializeDebuggerEditor (...)` – inicializuje nastavení editoru na práci s monitorovaným procesem.
- `BreakpointEvent (...)` – událost informující editor o aktivaci breakpointu.
- `ProcessResumed (...)` – událost informující editor o pokračování procesu / vlákna.
- `BreakPointToggle (...)` – událost informující monitorovací instanci o nastavení breakpointu v editoru.
- `FillMemory ()` – vyplní editor paměti,
- `FillDASM ()` – vyplní disassembler,
- `FillStack ()` – vyplní data zásobníku,
- `FillContext ()` – vyplní kontext a
- `FillFlags ()` – vyplní flagy.

B.3. Moduly

B.3.1. mdlAbstraction

Modul obsluhující spuštění Monitorovací aplikace, uchovává seznam všech vytvořených monitorovacích instancí.

- `Main ()` – počáteční funkce, instaluje knihovny (pokud je potřeba), nastavuje potřebná privilegia procesu, vytváří hlavní okno `frmMain` a nastavuje subclassování okna `frmSubClass`.
- `NewMonitor (...)` – vytvoří novou instanci `clsMonitor`.
- `GetMonitorByMonitoredPID (...)` – k PID přiřadí obsluhující instanci `clsMonitor`.
- `LoadPlugin (...)` – načte zadaný plugin.

B.3.2. mdlBrowser

Obsahuje funkce pro vyvolávání dialogových oken a další.

- `Dialog_VyberAdresar (...)` – vyvolá dialog pro výběr adresářů.
- `Dialog_OtevriSoubor (...)` – vyvolává dialog pro otevření souboru.
- `Dialog_UlozSoubor (...)` – vyvolává dialog pro uložení souboru.
- `GetIcon (...)` – získá ikonu zadaného spustitelného souboru.

B.3.3. mdlDeclares

Modul obsahuje pouze deklaraci API funkcí, struktur a konstant.

B.3.4. mdlFiles

Obsahuje základní funkce pro práci se soubory.

B.3.5. mdlHelpFunctions

Obsahuje různé pomocné funkce pro práci s řetězci, čísly a jiné.

- `ErrDescription (...)` – získá ze zadaného kódu chyby její slovní popis.
- `StrHexDumpToByteArray (...)` – konvertuje hex-dump do bytového pole.
- `FitsToFrame (...)` – testuje, zda zadané číslo náleží do zadaného intervalu.
- `TVToFile (...)` – konvertuje obsah prvku treeview do formátovaného textu.
- `AnyChangeIn...Array (...)` – testuje, zda jsou dvě pole rozdílná.

B.3.6. mdlHeuristicAndScriptShell

Obsahuje seznam zájmových skupin API funkcí.

B.3.7. mdlProcess

Modul obsahující všechny nezbytné funkce pro práci s procesy.

- `Process_Run (...)` – vytvoří nový proces ze zadaného souboru.
- `Process_InjectDLL (...)` – nainjektuje (načte) zadanou dll knihovnu do zadaného procesu.
- `Process_EnumProcesses (...)` – vytvoří seznam všech aktuálně běžících procesů.
- `Process_EnumModulesSoft (...)` – vytvoří seznam všech načtených modulů v zadaném procesu.
- `Process_EnumIAT (...)` – vytvoří seznam všech položek v IAT zadaného modulu v zadaném procesu.
- `Process_EnumEAT (...)` – vytvoří seznam všech položek v EAT zadaného modulu v zadaném procesu.
- `Process_EnumImageLocalFunctions (...)` – pokusí se pomocí skenování kódu spustitelné aplikace odhadnout lokální funkce.
- `Process_EnumThread (...)` – vytvoří seznam všech aktuálně existujících vláken v zadaném procesu.
- `Process_IsSuspended (...)` – zjistí, zda jsou všechna vlákna v zadaném procesu pozastavena.
- `Thread_Suspend (...)` – pozastaví zadané vlákno.
- `Thread_Resume (...)` – uvolní zadané vlákno.
- `Process_GetExecutableImage (...)` – získá cestu ke spustitelnému souboru zadaného procesu.
- `Process_GetAddressDetail (...)` – získá detaily k zadané adrese v kontextu zadaného procesu (např. překlad adresy na jméno funkce a modul).
- `IsProcessValidMem (...)` – testuje, zda je zadaný pointer validní v zadaném procesu.
- `GetHandleInfo (...)` – získává dodatečné informace k zadanému handlu.
- `GetNTHeader (...)` – získává strukturu `IMAGE_NT_HEADERS` ze zadaného modulu v zadaném procesu.
- `GetProcAddressEx (...)` – překládá jméno funkce a knihovny na adresu v kontextu zadaného procesu.

B.3.8. mdlSubClass

Zpracovává přijaté zprávy ze subclassovaného okna `frmSubclassForm`.

- `SubClassForm (...)` – zapíná / vypíná subclassování.
- `WindowProc (...)` – rutina zpracovávající přijaté WM zprávy.
- `ProcessMonitorCall (...)` – funkce zpracovávající WM zprávy typu `WM_MONITOR_CALL_HANDLER`.
- `ProcessMonitorReturnAddress (...)` – funkce zpracovávající WM zprávy typu `WM_MONITOR_RETURN_HANDLER`.

C. Obsah přiloženého CD

V samotném závěru práce je uveden stručný popis obsahu přiloženého CD, tj. závazné adresářové struktury, důležitých souborů apod.

`bin/`
: LDebugger.exe spustitelné přímo z CD. Adresář obsahuje i všechny potřebné knihovny a další soubory pro bezproblémové spuštění programu.

`doc/`
: Dokumentace práce ve formátu PDF, vytvořená dle závazného stylu KI PřF pro diplomové práce, včetně všech příloh, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru dokumentace (v ZIP archivu), tj. zdrojový text dokumentace, vložené obrázky, apod.

`src/`
: Kompletní zdrojové texty programu LDebugger se všemi potřebnými (převzatými) zdrojovými texty, knihovnami a dalšími soubory pro bezproblémové vytvoření spustitelných verzí programu (v ZIP archivu).

`readme.txt`
: Instrukce pro instalaci a spuštění programu LDebugger, včetně požadavků pro jeho provoz.

Navíc CD obsahuje:

`data/`
: Ukázková a testovací data použitá v práci a pro potřeby obhajoby práce.

`install/`
: Instalátory aplikací, knihoven a jiných souborů nutných pro provoz programu.

U veškerých odjinud převzatých materiálů obsažených na CD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro materiály, u kterých toto není splněno, je uveden jejich zdroj (webová adresa) v textu dokumentace práce nebo v souboru `readme.txt`.