МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРИЗОВАНИХ СИСТЕМ ЗАХИСТУ ІНФОРМАЦІЇ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
_____С.В.Казмірчук
«_____»_____2020 р.

На правах рукопису
УДК

МАГІСТЕРСЬКА АТЕСТАЦІЙНА РОБОТА

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
«МАГІСТР»

Тема: Програмний засіб захисту ОС Windows на базі сучасних технологій Intel

Автор: О.Б. Кочмар

Науковий керівник: к.т.н., доц. А.Б. Єлізаров

Нормоконтролер: к.т.н., доц. А.Б. Єлізаров

Київ 2020

ВСТУП

У нинішній час постійного використання персональних комп'ютерів як у робочих, так і особистих цілях, надзвичайно важливим є вберегти їх дані від несанкціонованого доступу та його наслідків, адже це впливає не лише на коректність роботи системи, а й на особисту конфіденційність користувачів. Задля забезпечення захисту необхідно ретельно слідкувати за файлами та програмами, які потрапляють до операційної системи. Суміжним важливим аспектом є також моніторинг поведінки таких файлів та додатків у системі та відповідна реакція у разі виявлення несанкціонованих процесів. Це може бути забезпечено шляхом програмного впровадження таких сучасних технологій корпорації Intel, як SMEP, SMAP, XDB, а також PaX, DEP, ASLR.

Актуальність роботи полягає у забезпеченні захисту операційної системи MS Windows та її даних від несанкціонованого доступу за допомогою досліджуваних сучасних технологій Intel. Це може стати в нагоді не лише пересічним користувачам, але й розробникам програмного забезпечення стосовно можливості подальшого вдосконалення забезпечення захисту операційних систем.

Метою роботи є удосконалення програмного засобу захисту операційної системи MS Windows на базі досліджуваних сучасних технологій Intel. Досягнення мети потребувало вирішення наступних **завдань**:

– дослідження сучасних технологій Intel, які задовольняють вимоги забезпечення захисту виконуваного простору, у тому числі принципу їх дії, реалізації та підтримки, а також супутніх понять;

– удосконалення програмного засіб захисту ОС MS Windows на основі Intel SMAP шляхом впровадження механізмів захисту досліджуваних технологій;

– тестування та оцінку ефективності впроваджених удосконалень.

Галузь застосування. Розроблений програмний засіб відноситься до галузі інформаційної безпеки і може бути використаний для підвищення рівня захищеності ІКС, зокрема їх операційних систем.

Об'єкт дослідження – операційна система Windows.

Предмет дослідження – механізми захисту сучасних технологій, використовуваних Intel, – SMEP, SMAP, XDB, PaX, DEP, ASLR, – у середовищі операційної системи MS Windows.

Оцінка сучасного стану проблеми на основі вітчизняної та зарубіжної літератури. Розглянуті у даній роботі технології майже не були досліджені вітчизняними фахівцями, особливо на рівні літературних праць, через відносну новизну. Більшість матеріалів досліджень цих технологій та впливу їх роботи на систему – закордонного авторства. Окрім того, практичні дослідження роботи технологій з метою забезпечення захисту системи присвячені в основному реалізації цих технологій у середовищі Linux, оскільки з ним пов'язане повсякденне використання більшості з них. Так, робота SMAP і PaX була досліджена у Linux, FreeBSD, OpenBSD, NetBSD, XEN та інших [17, 18, 30, 32]. Біт XD спочатку також набув практичного застосування у Linux та UNIX-подібних ОС (NetBSD, OpenBSD, macOS), це ж стосується й технології ASLR.

Що стосується застосування у ОС MS Windows, то чимало досліджень – як вітчизняних, так і зарубіжних, – пов'язані з технологією Intel SMEP. Так, Шишкін А. із видання «Positive Research Center» зазначає не лише позитивний вплив впровадження технології у ОС Windows 8, але й можливе його руйнування зловмисниками, наводить приклад необхідних для цього методів, здійснює їх аналіз і надає відповідні рекомендації задля уникнення таких ситуацій [21].

Вплив як SMEP, так і SMAP на ОС Windows і аналіз її захищеності на основі їх активації як окремо, так і у поєднанні, був предметом розгляду науковця Д. Грасса у його статті [21]. Окрім того, дані технології розглядалися дослідниками у рамках 18-го Міжнародного симпозіуму з досліджень стосовно атак, вторгнень та захисту «RAID 2015» [20]. Думки дослідників збігаються на тому,

що дія технологій у поєднанні є значно кращою і надійнішою за дію лише будь-якої однієї з них.

Крім того, увага дослідників також була значно приділена ефективності реалізації у ОС Windows технологій XDB, DEP і ASLR, їх розвиток у середовищі даної ОС відбувався майже одночасно. Функції XD були вперше реалізовані на архітектурі x86, починаючи з Windows XP Service Pack 2 (2004 р.) і Windows Server 2003 Service Pack 1 (2005 р.). Вони забезпечували захист виконуваного простору – тобто «запобігання виконання даних» (DEP). Такий захист за замовчуванням використовувався виключно для критичних служб Windows і надавався лише у разі підтримки біта процесорами x86.

Проте, фактично захист ОС все ще знаходився під питанням, оскільки реалізація DEP того часу не передбачала рандомізації компонування адресного простору (тобто підтримки ASLR), що, як виявили дослідники, дозволяло проводити потенційні атаки типу «повернення в бібліотеку», які можна було б дійсно використовувати для відключення DEP під час атаки [16]. Функціональність ASLR вперше з'явилася у Microsoft Windows у версіях Vista і Windows Server 2008 із реалізацією DEP за рахунок автоматичного використання PAE ядра в 32-бітній Windows і, власне, підтримки 64-бітних ядер.

Однак сьогодні цим технологіям можна довіряти щодо захисту від більшості поширених атак: у офіційному звіті Microsoft Security Intelligence дослідно підтверджено, що застосування ASLR та DEP дає підвищений захист системи від вірусів та інших численних несанкціонованих вторгнень [29].

Методи дослідження: тестування, порівняння, аналіз і синтез, спостереження, емпіричний метод.

Новизна одержаних результатів полягає у тому, що для ОС Windows вперше було розроблено програмний засіб на основі сукупності досліджуваних технологій у якості стороннього програмного засобу, а не модуля ядра, що дало можливість вибіркового управління захисними механізмами операційної системи з боку користувача.

Практична цінність полягає у тому, що завдяки розробленому засобу користувач має можливість програмно керувати апаратними технологіями Intel шляхом взаємодії з ядром операційної системи Windows та мікропроцесором, для чого зазвичай необхідно мати привілейований доступ до ресурсів системи. Користувачеві також надається можливість активувати захист операційної системи як за допомогою сукупності технологій, так і шляхом обрання лише окремих з них. Програмний засіб на практиці захищає вказану операційну систему, а отже і дані її користувачів, від несанкціонованого доступу, що підтверджено експериментально із залученням тестування системи щодо реакції на атаки. Окрім цього, дана робота містить значну кількість теоретичного матеріалу стосовно обраної теми, що може бути використано з навчальною метою.

Апробація. Основні положення роботи доповідалися та обговорювалися на таких конференціях:

- XVI Міжнародна наукова конференція «Динаміка сучасної науки». Софія, 15-22 липня 2020.
- XVI Міжнародна науково-практична конференція «Наукова індустрія європейського континенту». Прага, Publishing House «Education and Science» s.r.o., 22-30 листопада 2020.

РОЗДІЛ 1. ТЕОРЕТИЧНА БАЗА

1.1. Сучасні технології Intel

Центральні процесори від виробника зі світовим ім’ям Intel чимдалі, тим все більше адаптують свої рішення під вимоги безпеки. Для пересічного користувача 21 сторіччя захист своїх ПК від зловмисників є нагальною потребою, адже його відсутність може не тільки нанести шкоди конфіденційним даним, але і самій системі, внаслідок чого постраждає її працездатність. Тож, захист усієї операційної системи є нагальною вимогою, адже він передбачає запобігання будь-якому небажаному втручанню ззовні.

Саме тому нагальним і досі відкритим запитанням є «як ядро системи може бути захищене від виконання сторонніх несанкціонованих процесів» (тобто процесів, які не були передбачені системою чи користувачем для виконання і можуть нанести шкоди системі). Застосування сучасного захисту ядра є обов'язковою умовою для запобігання різних типів експлуатації ОС через ядро та пам’ять. Сьогодні з метою вирішення даного питання застосовується захист виконуваного простору, тобто простору пам’яті, у якій відбувається виконання програм чи файлів. Такий захист полягає у забороні взаємодії з ненадійними файлами та програмами у певних ділянках пам’яті, позначаючи їх невиконуваними. Тобто, задля забезпечення захисту виконуваного простору важливо «вберегти» ядро системи від будь-яких можливих несанкціонованих впливів з боку пам’яті користувача. Окрім цього, важливо не забувати про протилежне – захист простору пам’яті користувача від ядра.

Для впровадження такого захисту на практиці корпорація Intel пропонує свої технології – SMEP, SMAP, XDB. На сьогоднішній день з тією ж метою застосовуються також технології PaX, DEP, ASLR. Принцип дії кожної з запропонованих технологій розглядається у наступних підрозділах.

1.1.1. XDB

XDB або XD-Bit – це технологія Intel, яка використовується процесорами для розділення областей пам’яті з метою зберігання інструкцій процесора (тобто коду) або даних. Вона, фактично, є перейменованим атрибутом NX-Bit – бітом заборони виконання коду з області невиконуваних сторінок пам’яті, який вперше був застосований для захисту системи від помилок програм. Задля роботи даної технології необхідна не лише апаратна підтримка біта з боку процесора, але й з боку ядра ОС.

Процесори x86, починаючи з 80286, включали аналогічну функціональну можливість, реалізовану на рівні сегмента. Однак майже всі операційні системи для версій x86 мали однорівневу (несегментовану) модель пам’яті, тому вони фактично не мали змоги використовувати цю апаратну технологію. Пізніше AMD додала біт «no-execute» (NX) до елементу таблиці сторінок віртуальної пам’яті у рамках архітектури AMD64, забезпечуючи механізм, який може контролювати посторінкове виконання замість посегментного. Після рішення AMD включити цей функціонал у свій набір інструкцій, Intel аналогічно реалізувала XD-bit в процесорах x86, починаючи з процесорів Pentium 4, заснованих на більш пізніх ітераціях ядра Prescott [23].

Сучасні програми чітко поділяють на сегменти коду, даних, неініційованих даних, а також динамічно-розподілену область пам’яті, яка поділяється на купу і програмний стек. Якщо програма написана без помилок, покажчик команд ніколи не вийде за межі сегментів коду, однак, в результаті програмних помилок управління може бути передано до інших областей пам’яті. При цьому процесор перестане виконувати запрограмовані дії, натомість виконуватиме випадкову послідовність команд, за які він прийматиме дані, що зберігаються в цих областях. Це триватиме доти, доки процесор не «зустріне» неприпустиму послідовність, чи не спробує виконати операцію, що порушує цілісність системи, яка викличе реакцію системи захисту. У обох випадках програма завершиться аварійно. Також процесор може зустріти послідовність, інтерпретовану як команди переходу до вже пройденої адреси. В такому випадку процесор

увійде в нескінченний цикл, і програма «зависне», забравши 100% процесорного часу. Для запобігання подібних випадків був введений вказаний додатковий атрибут XD: якщо деяка ділянка пам’яті не призначена для зберігання програмного коду, то всі її сторінки повинні позначатися цим бітом і у разі спроби передати туди управління, процесор сформує виняток – і ОС миттю аварійно завершить програму, подавши сигнал виходу за межі сегмента (SIGSEGV).

Окрім того, досить часто такі помилки використовувалися зловмисниками для несанкціонованого доступу до комп’ютерів і написання вірусів, що використовують вразливості різних типів. Завдяки атрибуту XD (NX) отримання несанкціонованого доступу шляхом, наприклад, використання атак типу «переповнення буфера» стає неможливим. Це досягається шляхом позначення ділянки стека бітом, у результаті чого будь-яке виконання коду в ньому заборонено. Тепер, якщо передати управління стеку, спрацює захист. Хоч програму і можна змусити аварійно завершитися, але використовувати її для виконання довільного коду стає досить складно (для цього потрібно помилкове зняття програмою NX-захисту).

XD-біт є найстаршим розрядом елемента 64-бітних таблиць сторінок, використовуваних процесором для розподілу пам'яті в адресному просторі. 64-розрядні таблиці сторінок використовуються операційними системами, що працюють в 64-бітному режимі (IA-32e), або у 32-розрядному режимі з включеним режимом розширення фізичних адрес (PAE).

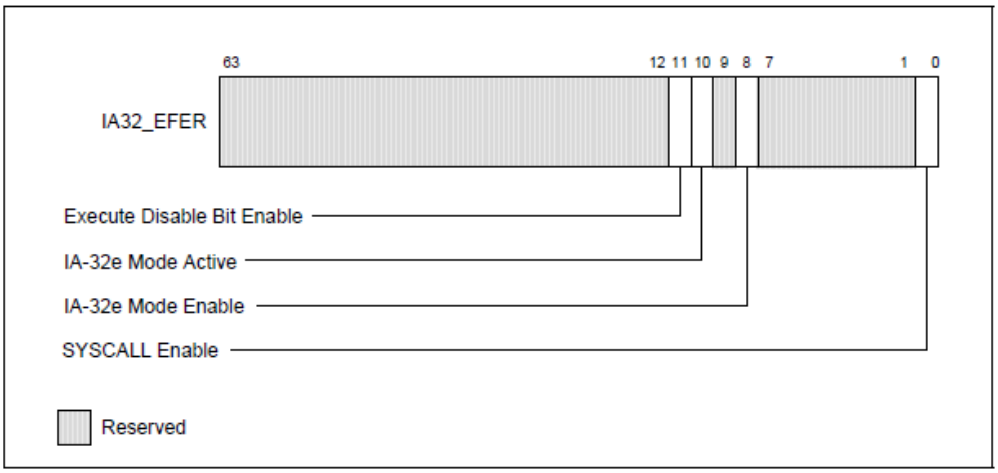
Якщо XD підтримується апаратно, технології, активація яких залежать від цього біта, є підтримуваними та активованими за замовчуванням.

1.1.1.1. Конфігурація XDB.

Процесори з підтримкою даної технології Intel позначаються літерою "J" після номера моделі процесора.

Програмно можна визначити присутність підтримки біта, використовуючи CPUID-інструкцію. CPUID.80000001H:EDX.NX [bit 20] = 1 означає, що біт підтримується. У такому разі, програмна активація можлива за допомогою додаткового контрольного регістра IA32_EFER MSR в x86-64 серії процесорів, а

саме його 11-го біта, що вмикається установкою IA32_EFER.NXE[bit 11] = 1 (див. рис. 1.1). Якщо ж XDB не підтримується, то запис на установку IA32_EFER.NXE викликає # GP виняток. Існуючі механізми захисту пам'яті продовжують працювати незалежно від установки біта [24].



63:12	11	10	9	8	7:1	0
Reserved	Execute-disable bit enable (NXE)	IA-32e mode active (LMA)	Reserved	IA-32e mode enable (LME)	Reserved	SysCall enable (SCE)

Рис. 1.1 Макет IA32_EFER MSR

XDB у сторінкових структурах покращує захист сторінок даних. Процесорні інструкції не можуть бути завантажені з пам'яті, якщо IA32_EFER.NXE = = 1 і XDB встановлений у будь-якої сторінкової структури, що використовується для відображення сторінки. На рис. 1.2 наведено опис дозволеного використання сторінки в залежності від значення біта заборони виконання (біт 63) відповідного запису на кожному рівні сторінкових структур. Заборона виконання може бути активована використовуючи XDB на будь-якому рівні сторінкових структур, незалежно від значення відповідного біта на інших рівнях. Коли біт заборони виконання не активований, сторінка може бути використана і як сторінка даних, і як кодова.

Execute Disable Bit Value (Bit 63)				Valid Usage
PML4	PDP	PDE	PTE	
Bit 63 = 1	*	*	*	Data
*	Bit 63 = 1	*	*	Data
*	*	Bit 63 = 1	*	Data
*	*	*	Bit 63 = 1	Data
Bit 63 = 0	Bit 63 = 0	Bit 63 = 0	Bit 63 = 0	Data/Code

Рис. 1.2 Тип сторінкового захисту у режимі IA-32e у залежності від значення біта заборони виконання

1.1.2. PaX

Однією з технологій забезпечення захисту виконуваного простору, робота якої напряму пов'язана із бітом XD, є PaX. Вона може емулювати функціональність біта або безпосередньо використовувати його за наявності його апаратної підтримки у системі.

Технологія PaX сьогодні використовується здебільшого для ОС Linux. Вона забезпечує кілька типів захисту, таких як: захист пам'яті, захист шляхом управління доступом на основі ролей, захист файлової системи, аудит ядра, захист виконуваних файлів, захист мережі, фізичний захист, а також можливість ведення журналу (протоколювання). Забезпечення захисту здійснюється за рахунок спостереження за поведінкою системи та її додатків, що дозволяє вчасно виявляти вразливості та попереджувати, стримувати їх прояв, замість того, щоб захищати систему вже від дії експлойтів. Найбільш суттєвим захистом, що забезпечується PaX, є захист ядра та пам'яті, яка в більшості випадків може запобігти ескалації привілеїв, експлуатації пам'яті та вберегти систему від наслідків виконання незахищених програм.

Тож, метою запровадження PaX є дослідження різних механізмів захисту від використання програмних помилок, які дають зловмиснику довільний доступ на читання / запис до адресного простору процесів. PaX дозволяє налаштувати мінімальні права доступу до сторінок пам'яті для виконуваних файлів, що у результаті дає їм можливість виконувати лише необхідні користувачеві дії, зумовлені їх функціональністю. Технологія позначає сегмент даних програм в пам'яті як недоступний для виконання (тому, що він не може містити програмних директив, які необхідно виконати), а сегмент коду – як той, що не перезаписується, і, на додачу до цього, при кожному запиті виконує рандомізацію сторінок пам'яті, тобто виділяє пам'ять програмі з довільних ділянок. Це надає можливість попередити і не допустити позначення області розміщення коду як такої, що доступна для запису, а області даних – як виконуваної.

1.1.2.1. Реалізація та конфігурація технології PaX.

Надання захисту технологією відбувається за допомогою використання біта NX процесора у інструкціях PAGEEXEC і SEGMEXEC, та використання обмеження захисту сторінок (MPROTECT).

Функція mprotect()відповідає за налаштування прав доступу до областей пам’яті у складі технології, адже PaX повинна гарантувати відсутність одночасно записуваних та виконуваних ділянок оперативної пам’яті. Активація захисту виконуваного простору технологією PaX, включно з обмеженнями функції mprotect(), гарантує, що ніякі зіставлення пам’яті не будуть позначені виконуваними після можливості зміни їх вихідного стану. У результаті стає неможливим виконання ділянок пам’яті під час і після запису до них доти, доки ці області не будуть знищені, а отже, код (розміщений в цих ділянках) не може бути інтегрований у будь-який додаток (шкідливий чи ні) з внутрішнього або зовнішнього джерела. Це гарантує, що сторінки пам’яті не стануть сприятливим середовищем для поширення атак. Із включеними обмеженнями mprotect() програма більше не може порушувати політику невиконуваних сторінок, яку PaX спочатку встановлює для всіх виділень пам'яті; таким чином, функція mprotect() може розглядатися як суворе дотримання політики безпеки, в той час як «примусово за допомогою NX невиконувані сторінки» без цих обмежень можуть розглядатися як більш вільна форма застосування.

Однак, така перевага технології водночас створює проблему для додатків, таких як JIT-компілятори Java, які повинні генерувати код в якості базової функції під час виконання, оскільки наслідком роботи технології є те, що програми не можуть самі виконувати дані, які вони створили як програмний код. Задля обходу даної незручності системний адміністратор може позначити виконуваний файл програми як такий, до якого не застосовуватимуться ці обмеження.

Функція mmap() у складі технології використовується або для відображення загальної пам’яті, або для завантаження спільних бібліотек. Через це

вона має надавати записувану або виконувану ОП, в залежності від умов використання. Поточна реалізація PaX надає доступні для запису анонімні зіставлення пам’яті. Зіставлення пам’яті з підтримкою файлів стають доступними для запису тільки в тому випадку, якщо виклик mmap() вказує дозвіл на запис. Функція mmap() ніколи не буде повертати зіставлення, які є як записуваними, так і виконуваними, навіть якщо ці дозволи явно запитуються у виклику.

Функція інструкції PAGEEXEC полягає у позначенні сторінок невиконуваними за допомогою використання логіки сторінкової організації процесорів на базі IA-32. Інструкція використовує біт NX (який безпосередньо дозволяє позначати сторінки пам’яті як виконувані або невиконувані) або емулює його програмно у разі, якщо процесор не підтримує апаратний NX. Це дещо знижує продуктивність системи, але значно підвищує її безпеку. У такому разі кожній сторінці пам’яті шляхом зміни рівня доступу для невиконуваних сторінок присвоюється емульований біт, що є можливим завдяки використанню буфера асоціативної трансляції (TLB), який у процесорів Intel розділений на два незалежні буфера в залежності від типу доступу: виклики інструкцій для виконання коду кешуються до ITLB, виклики для читання/запису – до DTLB (якщо для сторінки здійснюється виклик доступу обох типів, то обидва буфери TLB матимуть запис). Роль TLB полягає в тому, щоб зберігати кеш для трансляцій віртуальних або фізичних адрес, які процесор повинен обробляти щоразу при виклику доступу до пам’яті. Без TLB процесору довелося б виконувати перегляд таблиці сторінок для кожного такого доступу до пам’яті, що звичайно ж, було б згубно для продуктивності.

TLB працює наступним чином: щоразу, коли процесор хоче отримати доступ до заданої віртуальної адреси, він спочатку перевіряє, чи є у TLB збережена у кеш трансляція даної адреси. При наявності такої, процесор отримуватиме фізичну адресу безпосередньо з запису у TLB, в іншому випадку він буде виконувати перегляд таблиці сторінок, щоб знайти необхідний запис і кешувати результат в TLB. Так, як біти супервізора перевантажені для представлення NX (для обробки сторінок NX в сегменті коду), то у випадку спроби отримання до-

ступу до сторінки, кеш якої відсутній і у буфері асоціативної трансляції, і у таблиці сторінок, або ж він не співпадатиме за типом з викликом, ОС генерує помилку сторінки. Така помилка також дозволяє PaX визначити, чи намагалася програма виконати сторінку у вигляді коду. Якщо технологія виявляє помилку ITLB, то процес завершується; в іншому випадку ОС (Linux) дозволяє завантаження DTLB, і виконання триває в звичайному режимі.

Якщо апаратно біт NX підтримується, інструкція використовуватиме його, не вимагаючи жодних витрат на продуктивність.

Перевагою PAGEEXEC є те, що вона не ділить адресний простір пам’яті навпіл, а отже, не впливає на продуктивність. Однак для емуляції вона працює повільніше, ніж SEGMEXEC, і в деяких випадках призводить до серйозного зниження продуктивності.

SEGMEXEC емулює функціональність біта NX на процесорах IA-32 (x86), з якими ОС працює в захищеному режимі з можливістю мапування (також зіставлення – тобто розбиття простору пам’яті на сторінки). Це означає, що для кожного доступу до пам’яті (неважливо, шляхом виклику інструкцій або звичайного доступу до даних) процесор виконуватиме двоетапну трансляцію адрес. На першому етапі логічна адреса, декодована з інструкції, транслюється в лінійну (віртуальну) за допомогою логіки сегментації, що дає можливість налаштувати сегменти, які дозволяють реалізувати невиконання сторінок.

Ідея полягає в розділенні лінійного адресного простору користувача об’ємом 3 ГБ на дві рівні половини і використанні однієї для зберігання мапувань, призначених для доступу до даних (через визначення дескриптора сегмента даних, щоб охопити лінійний діапазон адрес 0-1,5Гб), а іншої – для зберігання мапувань, призначених для виконання (через визначення дескриптора сегмента коду, щоб охопити лінійний діапазон адрес 1,5-3 ГБ). Оскільки виконувані мапування можуть бути використані й для доступу до даних, то необхідно переконатися, що вони відображаються в обох сегментах і віддзеркалюють один одного. Далі відбувається відокремлення викликів доступу до даних від викликів інструкцій таким чином, щоб вони займали різні лінійні адреси і, от-

же, дозволяли здійснювати управління залежно від типу доступу. Зокрема, якщо мапування, призначені для доступу до даних, присутні тільки в лінійному діапазоні адрес 0-1,5 ГБ, то виклик інструкцій за тими ж логічними адресами закінчиться в лінійному діапазоні адрес 1,5-3 ГБ і викличе помилку сторінки.

Тобто фактично, інструкція SEGMEXEC скорочує простір віртуальної пам’яті процесу вдвічі (див. додаток А). Однак це дійсно підвищує продуктивність у випадку емуляції на архітектурах IA-32 (x86). Так, як відображення стосуються одних і тих самих сторінок фізичної пам’яті, то інструкція не створює подвійного навантаження на систему за рахунок використання оперативної пам'яті.

Одним із компонентів технології є інструкція NOEXEC. Мета її застосування полягає у запобіганні впровадження і виконання коду в адресному просторі процесу і анулюванні такого виду експлойта. Перша функція, яку реалізує NOEXEC, – це виконувана семантика (executable semantics on) на сторінках пам'яті. Вона відіграє важливу роль, оскільки дозволяє розділити властивості запису та виконання на сторінках пам'яті (у більшості систем, сторінка, що є доступною для запису, також доступна і для читання, а отже, і для виконання також). Це, в свою чергу, робить можливим ряд вторгнень, які в іншому випадку було б легко усунути.

Наступна особливість NOEXEC полягає в тому, що ядро фактично використовує нарешті доступну виконувану семантику. Зокрема, PaX робить стек і купу (всі анонімні зіставлення у цілому) невиконуваними. Крім того, зіставлення виконуваних файлів створюються з необхідними точками доступу, тобто явно виконуваними будуть тільки ті сегменти таких файлів, що містять код для виконання.

Остання функція NOEXEC полягає у блокуванні дозволів на сторінках пам'яті. Це перешкоджає створенню мапування записуваних / виконуваних файлів (анонімні зіставлення вже зроблені невиконуваними), а також перетворенню записуваних або невиконуваних мапувань на виконувані та навпаки. Оскільки це блокування порушує роботу реальних додатків, яким дійсно необ-

хідно генерувати код під час виконання, PaX дозволяє послабити ці обмеження для кожного виконуваного файлу (поряд з іншими функціями).

1.1.3. DEP

Інша технологія, пов’язана з XDB, DEP, спрямована на фактичне «запобігання виконання даних» шляхом маркування необхідних ділянок пам’яті як невиконуваних, ділянок «тільки для даних». DEP запобігає запуску коду зі сторінок даних, таких як купа, стеки і пули пам'яті. Відповідно, у результаті виконання будь-якого коду, що запускається на виконання із захищеної сторінки даних будь-якої з наведених областей, буде примусово завершене через виняток порушення доступу до пам'яті, і відсутність його обробки. Таким чином, DEP допомагає запобігти атакам переповнення буфера, перехоплюючи їх спроби на здійснення і викликаючи виняток.

Як і у випадку з технологією PaX, для роботи DEP у 32-розрядній версії Windows (починаючи з Windows XP Service Pack 2) необхідна процесорна підтримка режиму розширення фізичних адрес (PAE). На 64-розрядній версії технологія працює без залучення додаткових зусиль.

Якщо програма намагається запустити код із захищеної сторінки, вона отримує виняток із кодом стану STATUS_ACCESS_VIOLATION. Якщо програма повинна запускати код зі сторінки пам'яті, вона повинна виділити і встановити відповідні атрибути захисту віртуальної пам'яті. Виділена пам'ять повинна бути позначена як PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE або PAGE_EXECUTE_WRITECOPY. Розподіл купи, що виконується шляхом виклику функцій malloc і HeapAlloc, не є виконуваним.

Як і PaX, дана технологія може працювати у двох режимах – апаратному та програмному. Перший режим орієнтований на процесори з підтримкою біта XD і передбачає можливість маркування сторінок невиконуваними, другий – використовується для процесорів без підтримки біта відповідно і забезпечує додаткову перевірку обробки винятків (див. додаток Б).

У разі апаратної підтримки біта XDB та DEP, остання технологія є активованою у системі за замовчуванням і позначає всі комірки пам'яті в процесі як невиконувані, якщо тільки ця комірка явно не містить виконуваний код. Оскільки кожна сторінка може бути індивідуально позначена як виконувана або невиконувана, то ОС Windows дозволяє програмам контролювати, які саме сторінки забороняють виконання через свій API, а також через заголовки розділів в PE-файлі. В API під час виконання доступ до біта XD надається через виклики інтерфейсу прикладного програмування (API) Win32 VirtualAlloc[Ex] і VirtualProtect[Ex], надаючи таким чином виділення виконуваної пам’яті з відповідними параметрами її захисту та заборону на запис до виділеної пам'яті відповідно. Заборона доступу на запис забезпечує максимальний захист виконуваних областей адресного простору процесу [28]. DEP функціонує посторінково, змінюючи значення біта для записів таблиці сторінок (PTE).

У режимі користувача виняток DEP призводить до STATUS_ACCESS_VIOLATION (0xc0000005) у системах Windows. Перший параметр ExceptionInformation, що містить структура EXCEPTION_RECORD, повертає тип події, яка порушила права доступу. Так, якщо ExceptionInformation[0] = 8, то даний параметр вказує на те, що порушення доступу було спричинене виконанням коду. Для більшості процесів режиму користувача виняток STATUS_ACCESS_VIOLATION зазвичай є необроблюваним і призводить до їх завершення.

DEP також застосовується до драйверів в режимі ядра. DEP для областей пам’яті в режимі ядра не може бути вибірково включений або відключений. У 32-розрядних версіях Windows DEP застосовується до стеку за замовчуванням. Натомість у режимі ядра 64-розрядних версій Windows DEP застосовується як до стеку, так і до пулів підкачки та сеансів. Драйвери пристроїв не можуть виконувати код зі стека, коли DEP увімкнена. Порушення доступу DEP в режимі ядра призведе до помилки перевірки 0xFC: ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY

Програмно-емульований DEP виконує додаткові перевірки механізмів обробки винятків в Windows. Емуляція технології відбувається за допомогою «безпечної структурованої обробки винятків» Microsoft – т. зв. SafeSEH, яка перевіряє, щоб при виникненні винятку під час виконання програми обробник винятків визначався додатком в тому вигляді, в якому був скомпільований спочатку. Якщо файли образів програми побудовані з використанням безпечної структурованої обробки винятків (SafeSEH), то DEP гарантує, що перед вивантаженням винятку обробник буде зареєстрований в таблиці функцій, розташованої у файлі образу. У протилежному випадку DEP гарантує, що перед вивантаженням винятку обробник винятків буде розташований в області пам'яті, позначеної виконуваною. Ефект цього захисту полягає в тому, що зловмисник не може додати свій власний обробник, який він зберіг на сторінці даних через неперевірене введення програми [25]. Хоча завершення процесу або збій системи як результат перевірки помилок – не ідеальний варіант, все ж це ефективна міра боротьби зі шкідливим кодом. Запобігання виконання шкідливого коду в системі може запобігти його поширенню або пошкодженню системи, шкідливі наслідки чого можуть легко перевищити небажані наслідки завершення виконуваного процесу.

Іншою перевагою застосування DEP є можливість розробників уникати виконання коду зі сторінок даних без явного маркування їх виконуваними.

Передбачається, що поведінка деяких додатків може бути несумісною із DEP. Так, це стосується програм, які виконують динамічну генерацію коду (наприклад, just-in-time (JIT)) і явно не позначають згенерований код дозволом на виконання, а також тих, що не створені з SafeSEH. Останні повинні мати свої обробники винятків, розташовані у виконуваних областях пам'яті. Проблеми сумісності драйверів із DEP в основному зумовлені використанням PAE. Загалом ці проблеми вирішуються виконанням вимог технології до програм.

1.1.3.1. Конфігурація технології DEP.

У Windows конфігурація функціонування технології може бути здійснена за допомогою змінних файлу Boot.ini або панелі управління у режимі адміністр-

ратора. Дана ОС підтримує чотири загальносистемні варіанти конфігурації апаратної та програмної реалізації DEP:

- Підключення: DEP за замовчуванням увімкнена для окремих системних двійкових файлів на процесорах з апаратною підтримкою технології; є можливість увімкнення DEP для сторонніх програмних додатків. Якщо для загальносистемної політики DEP задано дане значення, то одні і ті ж двійкові файли і додатки ядра Windows будуть захищені технологією як апаратно, так і програмно. Якщо система не підтримує апаратне забезпечення DEP, то основні двійкові файли і додатки будуть захищені лише програмно.
- Відключення: DEP увімкнена за замовчуванням для всіх процесів; користувач може вручну створити список конкретних додатків, до яких DEP не застосовується, а розробники та незалежні постачальники програмного забезпечення (ISV) можуть використовувати Application Compatibility Toolkit, щоб вимкнути захист DEP для одного або декількох додатків. Аналогічно попередній опції, якщо встановлена дана, то обрані додатки будуть звільнені як від апаратного захисту DEP, так і програмного.
- Завжди увімкнено: дія DEP поширюється для всієї ОС і всіх її процесів, запущених після увімкнення технології; підтримка режиму «Відключення» для певних додатків відсутня.
- Завжди вимкнено: дія DEP не поширюється на ОС взагалі, незалежно від апаратної підтримки; процесор може працювати в режимі PAE у 32-розрядних версіях Windows, якщо тільки параметр / NOPAE присутній у завантажувальному записі.

Налаштування DEP також можливе при завантаженні системи через BIOS відповідно до параметра політики захисту сторінок no-execute. Додаток може отримати поточний параметр політики, викликавши функцію GetSystemDEPPolicy. Залежно від параметра політики, додаток може змінити параметр DEP для поточного процесу, викликавши функцію SetProcessDEPPolicy.

1.1.4. ASLR

Сучасна реалізація технологій PaX і DEP неможлива без впровадження ASLR – технології «рандомізації розміщення адресного простору». Як і попередні, вона захищає від атак типу «переповнення буфера», а також «повернення в бібліотеку». ASLR випадковим чином упорядковує в адресному просторі позиції ключових даних процесів, у тому числі образу виконуваного файлу і позиції стека, купи і бібліотек (див. додаток В) [31].

Основна ідея цього підходу до захисту виконуваного простору заснована на спостереженні, що на практиці більшість атак вимагають завчасного знання адрес для здійснення атак на процеси системи і на низькій ймовірності випадкового визначення правильного розташування області адресного простору зловмисником. Введення ентропії для адрес при створенні кожного нового процесу змусить атакуючого вгадувати або перебирати варіанти необхідних адрес, що зробить спроби атаки помітними, тому що кожна невдала спроба, швидше за все, призведе до помилки і не дасть очікуваного зловмисником результату. Це, у свою чергу, дасть можливість виявляти такі події та реагувати на них. Забезпечувана безпека у цьому випадку підвищується за рахунок збільшення ентропії, що відбувається або за рахунок збільшення обсягу області віртуальної пам'яті, в якій відбувається рандомізація, або за рахунок зменшення її періоду.

Вгадування адрес має місце, коли застосована до кожного атакованого процесу рандомізація змінюється непередбачуваним чином. Фактично, атакуючий не може нічого дізнатися про майбутні рандомізації і має однакові шанси на успіх при кожній спробі атаки. Перебирання відбувається, коли атакуючий може щось дізнатися про майбутні рандомізації і використати це знання для прорахування адрес для своїх атак.

Щоб кількісно оцінити твердження про ймовірність успішності або безуспішності атак, необхідно ввести декілька змінних:

- *Rs* – кількість бітів, випадково розташованих (рандомізованих) в області стека;
- *Rm* – кількість бітів, рандомізованих у області mmap();

- *Rx* – кількість бітів, рандомізованих в основній виконуваній області пам’яті;
- *Ls* – розташування найменш значущого рандомізованого біта стека;
- *Lm* – розташування найменш значущого рандомізованого біта в області mmap();
- *Lx* – розташування найменш значущого рандомізованого біта в основній виконуваній області пам’яті;
- *As* – кількість випадково атакованих за одну спробу бітів стека;
- *Am* – кількість випадково атакованих за одну спробу бітів mmap();
- *Ax* – кількість випадково атакованих за одну спробу бітів основної виконуваної області пам’яті.

Нехай для процесора i386 було визначено такі значення змінних:

- *Rs* = 24; *Rm* = 16; *Rx* = 16;
- *Ls* = 4; *Lm* = 12; *Lx* = 12.

Тобто адреси стека мають випадково розміщені 24 біти в позиціях 4-27, а чотири найменш і найбільш значущі біти залишаються без впливу рандомізації. У даній ситуації може бути атаковано більше одного біта одночасно (очевидно, $A \leq R$), наприклад, шляхом багаторазового дублювання навантаження атаки в пам'яті можна атакувати найменш значущі біти рандомізації.

Ймовірності успіху атак в межах *x* числа спроб можна визначити за наступними формулами (1.1 – для вгадування і 1.2 – для перебору відповідно):

$$Pg(x) = 1 - (1 - 2^{-N})^x,\ 0 \leq x; \quad (1.1)$$

$$Pb(x) = \frac{x}{2^N},\ 0 \leq x \leq 2^N; \quad (1.2)$$

Тут *N* – кількість шуканих рандомізованих бітів:

$$N = Rs - As + Rm - Am + Rx - Ax. \quad (1.3)$$

Виходячи з вищевикладеного, ймовірність успіху атаки залежить від того, скільки бітів було атаковано за одну спробу і кількості спроб (див. рис. 1.3, рис. 1.4):

Pg(x) | x														
N	1	4	16	64	256	2^10	2^14	2^18	2^20	2^24	2^32	2^40	2^56	2^64
1	0.50	0.94	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1
2	0.25	0.68	0.99	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1
4	0.06	0.23	0.64	0.98	~1	~1	~1	~1	~1	~1	~1	~1	~1	~1
8	~0	0.02	0.06	0.22	0.63	0.98	~1	~1	~1	~1	~1	~1	~1	~1
16	~0	~0	~0	~0	~0	0.02	0.22	0.98	~1	~1	~1	~1	~1	~1
24	~0	~0	~0	~0	~0	~0	~0	0.02	0.06	0.63	~1	~1	~1	~1
32	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.63	~1	~1	~1
40	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.63	~1	~1
56	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.63	~1

Рис. 1.3 Залежність ймовірності успішності атаки від кількості бітів, атакованих шляхом вгадування їх адрес, та кількості спроб

Pb(x) | x														
N	1	4	16	64	256	2^10	2^14	2^18	2^20	2^24	2^32	2^40	2^56	2^64
1	0.50													
2	0.25	1												
4	0.06	0.25	1											
8	~0	0.02	0.06	0.25	1									
16	~0	~0	~0	~0	~0	0.02	0.25							
24	~0	~0	~0	~0	~0	~0	~0	0.02	0.06	1				
32	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	1			
40	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	1		
56	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	~0	1	

Рис. 1.4 Залежність ймовірності успішності атаки від кількості бітів, атакованих шляхом перебирання їх адрес, та кількості спроб

Очевидно, що з точки зору захисту мета полягає в тому, щоб зробити N якомога вище, зберігаючи x якомога нижче. На жаль, N не контролюється стороною захисту (повторна рандомізація адресного простору під час виконання нездійсненна, оскільки частина необхідної інформації про переміщення просто втрачається), а скоріше залежить від природи помилки і навичок зловмисника. Зменшення N можливо, якщо зловмисник може зберігати кілька екземплярів навантаження атаки (наприклад, ланцюжок кадрів стека, якщо NOEXEC активний, або деякий шелл-код, якщо він не активний) в адресному просторі атакованого процесу. Зазвичай це можна зробити, використовуючи помилки типу

переповнення, коли зловмисник може заповнити безперервний діапазон пам'яті даними на свій вибір. Оскільки розмір цього діапазону пам'яті зростає вище значення *L*, що відноситься до даного діапазону, все більше і більше рандомізованих бітів можуть бути проігноровані в навантаженні атаки. Наприклад, щоб подолати всі *R* для заданого діапазону на i386, зловмиснику доведеться відправити 256 МБ даних, що не завжди є можливим.

З іншого боку, сторона захисту має досить певний контроль над значенням *x*: всякий раз, коли спроба атаки робить невірне припущення про рандомізовані біти, атакований процес / додаток переходить в стан, який призведе до збою системи, що обов'язково «помітить» ядро.

«Побічні ефекти» ASLR – це фрагментація адресного простору і виснаження пулу ентропії. Оскільки рандомізація зміщує цілі діапазони пам'яті, вона також буде випадково змінювати проміжки між ними (які раніше були постійними). Це, в свою чергу, змінить максимальний розмір розміщених там зіставлень пам'яті, а додатки, що створюють їх, зазнають невдачі. Нарешті, ASLR збільшує споживання пулу ентропії, так як кожен новий процес вимагає декілька випадкових бітів для визначення нового розміщення у адресному просторі. Однак залежно від моделі загроз системи дана реалізація може послабити вимоги до якості цієї ентропії.

Для використання ASLR виконувані файли потрібно збирати зі спеціальними прапорами. У результаті в коді не будуть використовуватися постійні адреси, але при цьому:

– збільшиться розмір коду виконуваних файлів;

– збільшиться час завантаження в пам'ять кожного виконуваного файлу;

– виникне додаткова несумісність з ПЗ і бібліотеками, розробленим під версії ОС без ASLR.

1.1.4.1. Реалізація технології ASLR.

Функціональність ASLR забезпечується декількома інструкціями у залежності від того, вплив на яку частину розподілу адресного простору є кінцевою метою:

RANDEXEC / RANDMMAP (використовує функцію delta_exec) - основний виконуваний код / дані / сегменти bss;

RANDEXEC / RANDMMAP (delta_exec) - керована пам'ять (купа);

RANDMMAP (delta_mmap) - mmap() керована пам'ять (бібліотеки, купа, стеки потоків, загальна пам'ять);

RANDUSTACK (delta_stack) – стек користувача;

RANDKSTACK - стек ядра (не входить в адресний простір процесу).

1.1.5. SMEP

Технологія SMEP була представлена із появою нового покоління процесорів Intel на базі архітектури Ivy Bridge. Вона пов’язана із захистом виконуваного простору у режимі супервізора (ядра). Цей режим визначається поточним рівнем привілеїв CPL < 3 і ядру надано необмежений доступ до адресного простору пам’яті (в той час як при CPL = 3 активним є режим користувача і користувачеві надано доступ лише до свого адресного простору).

Деякі операції неявно отримують доступ до структур системних даних з лінійними адресами. Одержуваний доступ до структур цих даних здійснюється у режимі супервізора незалежно від CPL. Приклади такого доступу включають: доступ до глобальної дескрипторної таблиці (GDT) або локальної дескрипторної таблиці (LDT) для завантаження сегментного дескриптора; доступ до дескрипторної таблиці переривань (IDT) при передачі переривання або винятку; доступ до сегмента стану завдання (TSS) як частина перемикання завдань або зміни CPL. Інші надання доступу, здійснені під час CPL < 3, називаються явним доступом режиму супервізора.

Права доступу також контролюються режимом лінійної адреси, як зазначено в записах структури сторінок пам’яті, які керують трансляцією лінійної адреси. Якщо прапор U/S (біт 2) має значення 0 принаймні в одному з записів структури сторінок пам’яті, то адреса належить адресному простору режиму супервізора. В іншому випадку адреса належить адресному простору режиму користувача.

• Для доступу режиму супервізора характерними є такі ознаки і можливості:

– Дані можуть бути прочитані (неявно або явно) з будь-якої адреси простору режиму супервізора.

– Також відбувається читання даних зі сторінок простору режиму користувача.

– Дані записуються до адресного простору ядра та простору користувача.

– Виклик інструкцій можливий з адрес як простору ядра, так і користувача.

Кожне з зазначених положень має свої нюанси і умови, які залежать від значень певних бітів керуючих регістрів.

Отже, технологія Intel SMEP полягає у запобіганні виконання коду, розміщеного на сторінці користувача, у режимі ядра, при поточному рівні привілеїв рівному 0. Тобто, технологія забезпечує захист ядра від експлойтів, пов’язаних з перевищенням привілеїв (експлойтів класу EoP – «Escalation of Privileges»).

Зазвичай при експлуатації вразливості у режимі ядра атакуючий виділяє буфер з шелл-кодом у режимі користувача й активує вразливість, отримуючи контроль над виконанням коду і перевизначаючи точку (адресу) виконання на вміст підготовленого буфера. При успішності експлуатації зловмисник отримує повний контроль над системою. SMEP чинить перешкоду, тому що в даних умовах місце для зберігання шелл-коду відсутнє. Атака, що не може бути здійснена через неможливість виконання шелл-коду, є безглуздою.

SMEP є частиною механізму захисту сторінок пам'яті. Дана технологія використовує вже існуючий прапорець у записі таблиці сторінок – U/S («User/Supervisor», 2-й біт), що вказує на те, сторінкою якого режиму є дана – режиму користувача або ядра. Власник сторінки визначає наявність доступу до даної сторінки пам'яті, тобто, якщо сторінка належить ядру операційної систе-

ми, яка виконує код в привілейованому режимі, то доступ до неї з користувацького додатка неможливий (див. додаток Г) .

1.1.5.1. Конфігурація технології SMEP.

Активація технології відбувається за допомогою 20-го біта керуючого регістра CR4 (див. рис. 1.5), який разом із іншими п’ятьма регістрами (CR0, CR1, CR2, CR3 і CR5-7) визначає режим роботи процесора і характеристики виконуваного в даний час процесу. CR4 містить групу прапорів, які включають кілька архітектурних розширень і вказують на підтримку певної ОС або підтримку виконання окремих можливостей процесора.

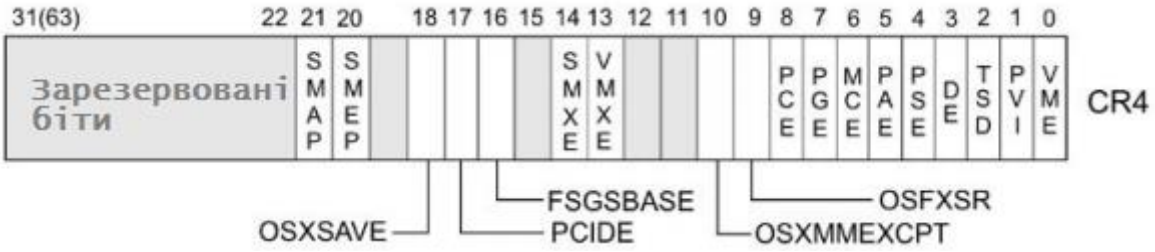


Рис. 1.5 Структура керуючого регістра CR4

Для маніпулювання бітами регістру використовуються інструкції MOV CRn. Префікси розміру операндів для цих інструкцій ігноруються. Встановлення 20-го біта CR4 у значення «1» змінює вплив прапорця U/S на доступ до сторінок пам'яті. Оскільки відомо, що біти 63:32 CR0 і CR4 зарезервовані і повинні бути заповнені нулями, то запис ненульового значення в будь-який з 32-х верхніх бітів призводить до обслуговування процесором винятку загального захисту (#GP, «General Protection Fault», #GP(0)) [24]. Проте на практиці при активності привілейованого режиму будь-яка спроба виконання коду, розташованого на сторінці користувача (у пам’яті певного додатку користувача; наприклад, адреси < 0x80000000 на 32-розрядній Windows з налаштуваннями за замовчуванням), призводить до апаратної генерації помилки сторінки («Page Fault», #PF) у результаті порушення прав доступу. Отже, ОС повинна здійснювати обробку порушення механізму SMEP за залученням обробника помилок сторінок. На ОС Windows дана ситуація викличе «bugcheck» і користувач побачить «синій екран смерті» (див. додаток Д).

Застосування інструкції MOV до значення CR4 для його зміни інвалідує всі записи TLB, у тому числі всі записи кешу сторінкових структур. Інвалідація – це оновлення елементів з метою забезпечення актуальності використовуваних даних. Однак, трапляються випадки, у яких програмне забезпечення надає перевагу ігноруванню інвалідації. Наприклад, коли CR4.SMEP = 0 і запис сторінкової структури модифіковано для зміни значення прапора U/S з 0 на 1. Тоді пропущена інвалідація може призводити до «випадкових» page-fault (#PF) винятків (пов’язаних з помилкою сторінки), наприклад, у відповідь на невдалу спробу доступу у режимі користувача, але без будь-яких інших несприятливих проявів. Такий виняток виникатиме максимум один раз для кожної з задіяних лінійних адрес.

Попередньо у даній роботі вже згадувалося про права доступу. Як саме значення SMEP-біта впливає на доступ до сторінок пам'яті? Зазвичай, воно є визначним лише для режиму супервізора і у першу чергу стосується надання прав для виклику інструкцій з адресного простору пам’яті.

- Якщо значення SMEP-біта керуючого регістра CR4 дорівнює 0 (CR4.SMEP = 0), то права доступу залежать від режиму сторінкової організації і значення прапора IA32_EFER.NXE:

– Для 32-розрядної сторінкової організації або якщо IA32_EFER.NXE = 0, то виклик інструкцій може здійснюватися з будь-якої адреси, для якої надано доступ у режимі користувача.

– Для пейджингу PAE або 4-рівневого пейджингу разом зі значення прапора IA32_EFER.NXE = 1 інструкції можуть бути зчитані з будь-якої адреси, доступної у режимі користувача з трансляцією, для чого значення прапора XD (біт 63) дорівнює 0 в кожному записі сторінкової структури, що керує трансляцією; і навпаки: інструкції не можуть бути викликані з будь-якої адреси користувацького режиму з трансляцією при тому, що значення біта XD дорівнює 1 в будь-якому записі сторінкової структури, що керує трансляцією.

- Якщо CR4.SMEP = 1, то ПЗ, що працює у режимі супервізора, не може викликати інструкції з будь-якої лінійної адреси, що доступна в режимі користувача.

Окрім цього, підтримка технології полягає у застосуванні інструкції CPUID процесора. За її допомогою можна контролювати підключення та відключення SMEP на апаратному рівні. Так, якщо при встановлених вхідних значеннях регістрів EAX=07H, ECX=0H інструкції CPUID вихідним значенням 7-го біта регістра EBX є 1 (EBX.SMEP [bit 7] = 1), то можна сказати, що біт керуючого регістра, який відповідає за стан технології (CR4.SMEP), встановлено на значення «1» і технологію запобігання виконанню режиму супервізора активовано. І навпаки, якщо при аналогічних встановлених вхідних значеннях тих саме регістрів CPUID вихідним значенням SMEP-біта є 0 (EBX.SMEP [bit 7] = 0), то CR4.SMEP=0 і технологію деактивовано [24].

64-розрядна версія Windows перевіряє підтримку SMEP при ініціалізації завантажувальних структур, заповнюючи змінну KeFeatureBits і яка надалі використовув при обробці помилок сторінок.:

KiSystemStartup()→ KiInitializeBootStructures() → KiSetFeatureBits()

На x86 версії вона вмикається також під час старту системи, у фазі «1» в функції KiInitMachineDependent() і потім ініціалізується для кожного ядра процесора, ініціюючи міжпроцесорне переривання, яке у результаті викликає функцію KiConfigureDynamicProcessor().

KiSystemStartup() → KiInitializeKernel() → KiGetFeatureBits()

Іншою частиною підтримки SMEP є код обробника помилок сторінки. Починаючи з версії ОС MS Windows 8 була додана нова функція – MI_CHECK_KERNEL_NOEXECUTE_FAULT(). Нею здійснюється перевірка порушення технологій SMEP і NX. Якщо відбулося порушення прав доступу, пов'язане зі SMEP або NX, то користувач бачить «синій екран смерті» з кодом помилки «ATTEMPTED_EXECUTE_OF_NOEXECUTE_ MEMORY»:

KiTrap0E() / KiPageFault() → MmAccessFault() → … →
MI_CHECK_KERNEL_NOEXECUTE_FAULT()

1.1.6. SMAP

SMEP хоча і ускладнює виконання атак, але все ж не гарантує повного її захисту. Тому дещо пізніше за SMEP у процесорах Intel архітектури Broadwell з метою підвищення безпеки і захисту від вразливостей, не усунених SMEP, була впроваджена технологія SMAP. Вона доповнює технологію запобігання виконанню режиму супервізора і запобігає запису в пам'ять та читання з неї коду, який несанкціоновано використовує режим супервізора, тоді як SMEP попереджає виконання цього коду.

Без технології запобігання доступу в режимі супервізора код, що виконується в режимі супервізора, зазвичай має повний доступ для читання і запису до відображень у пам'яті режиму користувача (або має повний доступ). Це робить систему вразливою перед експлойтами безпеки, у т.ч. експлойтами типу EoP, які змушують ядро незаплановано звертатися до пам'яті користувача. Технологія запобігає незапланованому доступу до простору даних користувача з коду ядра і є такою особливістю деяких процесорів, яка дозволяє програмам режиму супервізора додатково встановлювати такі зіставлення (мапування) у пам'яті користувача, доступ до яких викликає пастку. Спроба буда відхилена і виникне помилка сторінки (#PF).

Це ускладнює можливість обійти захист ядра за допомогою використання інструкцій або даних програми з пам’яті користувача. Крім того, SMAP може виявити помилковий/дефектний код ядра, який не відповідає передбаченим вимогам для доступу до пам'яті користувача (див. додаток Е) [24].

Стосовно можливих навантажень на систему з запровадженням і використанням технології, напевно відомо, що вона є [26]. Використання SMAP, окрім користі для операційної системи, може призвести до збільшення розміру ядра і уповільнення доступу до пам'яті простору користувача з боку коду режиму супервізора. Це пов’язано з тим, що SMAP повинна бути тимчасово відключена кожного разу, коли код режиму супервізора має намір отримати доступ до пам'яті користувача.

1.1.6.1. Конфігурація технології SMAP.

SMAP увімкнена при активній сторінковій організації пам'яті і встановленому 21-му біті керуючого регістра CR4 (див. рис. 1.5). Коли SMAP активовано, ПЗ, яке працює у режимі супервізора, не може звертатися до сторінок, які належать адресному простору користувача, як зазвичай; у такому разі воно просто не отримує доступ до даних, розміщених на лінійних адресах, які зазвичай доступні для програм, що працюють в режимі користувача. Таке обмеження дещо суперечить класичній ієрархії захисту пам'яті (при якій статус супервізора допускає будь-які види доступу), однак у деяких випадках допомагає протистояти діям шкідливих програм, а також спростити процедуру виявлення деяких помилок, що призводять до спотворення вмісту пам'яті.

Контролювати стан (активація та деактивація) SMAP на апаратному рівні можна за допомогою інструкції CPUID процесора. Якщо при встановлених вхідних значеннях регістрів EAX=07H, ECX=0H інструкції для регістра EBX є правдим, що EBX.SMAP [bit 20] = 1, то можна стверджувати, що технологію запобігання доступу в режимі супервізора активовано. Відповідно, коли при аналогічних вхідних значеннях тих саме регістрів EBX.SMAP [bit 20] = 0, то CR4.SMAP = 0 і технологія вимкнена [24].

Проте, постійне використання даної технології без жодного винятку зруйнувало б звичну роботу операційної системи. Оскільки у багатьох випадках ядро потребує доступ до сторінок адресного простору користувача, то задля узгодження таких ситуацій із роботою технології надається можливість перевизначення: технологію можна тимчасово відключити для доступу до пам'яті користувача з боку ядра. Так, як процесори здійснюють підтримку технології запобігання доступу до режиму супервізора через інструкції CPUID, то контролювати доступ ядра до простору пам’яті користувача можливо завдяки взаємодії з розширеним регістром EFLAGS.AC прапора AC. Для того, щоб легко встановити або зняти прапор, слугують інструкції STAC і CLAC відповідно (див. табл. 1.1). Це вимагатиме більш значних змін в коді ядра, ніж у випадку зі SMEP, але об’єм коду все ж залишається цілком керованим. Як і сама техноло-

гія, дані інструкції також створюють навантаження на систему і тому вимагають певний час для виконання.

Для структур системних даних, наприклад, таблиці дескрипторів, до яких безпосередньо звертається процесор, SMAP активна навіть при третьому рівні привілеїв (CPL = 3), незалежно від регістра EFLAGS.AC. Проте, коли біт SMAP в CR4 встановлений і явно орієнтована пам'ять зчитує та здійснює запис до сторінок пам’яті режиму користувача, що виконуються кодом, який працює з рівнем привілеїв менше третього, але регістр EFLAGS.AC не встановлений, то у результаті виникне помилка сторінки. Неявно орієнтоване читання і запис (наприклад, в таблиці дескрипторів) до сторінок пам’яті режиму користувача завжди викличе помилку сторінки, якщо SMAP включена, незалежно від значення EFLAGS.AC.

Виконання даних інструкцій та його наслідок є ідентичним як для 64-бітного режиму, так і 32-бітного.

Таблиця *1.1*

Інструкції технології SMAP

	STAC	**CLAC**
Функція	Активація SMAP	Деактивація SMAP
Формат коду інструкції	NP 0F 01 CB	0F 01 CA
Кодування операнду	ZO	NP
Алгоритм	EFLAGS.AC ← 1	EFLAGS.AC ← 0
Задіяні прапори	AC	

Отже, стан технології (активований чи навпаки) має вплив на права доступу в системі. Таким чином:

- Якщо CR4.SMAP = 0, то дані можуть зчитуватися з будь-якої адреси, доступної у режимі користувача, за допомогою ключа захисту, для якого доступ дозволений.

- Якщо CR4.SMAP = 1, то права доступу залежать від значення регістра EFLAGS.AC і від того, чи є доступ неявним або явним:

– Якщо EFLAGS.AC = 1 і доступ явний, то дані можуть бути прочитані з будь-якої адреси режиму користувача з ключем захисту, для якого дозволений доступ на читання.

– Якщо EFLAGS.AC = 0 або доступ неявний, то дані не можуть бути зчитані з жодної адреси користувацького режиму.

Деякі спроби виконати інструкції STAC і CLAC можуть завершитися невдало і призвести до помилок. Помилки невизначеності інструкцій мають позначення #UD і генерують обробку процесором винятків під назвою «Invalid Opcode Exception» – «Виняток помилкового коду». Вони належать до класу відмов і не зберігають код помилки.

Відмова – це виняток, який знаходиться і обслуговується процесором до виконання інструкції, що викликає помилку. Після обслуговування цього винятку управління повертається знову до інструкції (включаючи всі префікси), яка викликала відмову у своєму виконанні. Оскільки помилкова інструкція не виконується, то стан програми, що задіює її, не змінюється.

Також унаслідок виконання інструкції CLAC можуть виникнути винятки захищеного режиму («Protected Mode Exceptions»). Це може статися за умов, якщо:

– використовується префікс блокування шини LOCK;

– поточний рівень привілеїв є більшим за нульовий (CPL > 0);

– в інструкції CPUID з вхідними значеннями регістрів EAX=07H, ECX=0H, EBX.SMAP [bit 20] = 0, що говорить про деактивацію технології на апаратному рівні (CR4.SMAP=0).

Винятки режиму реальної адресації («Real-Address Mode Exceptions») зумовлені:

– використанням префікса блокування шини LOCK;

– тим, що при вхідних значеннях регістрів EAX=07H, ECX=0H вихідним значенням біта SMAP регістра EBX є 0 (EBX.SMAP [bit 20] = 0).

Можливі винятки режиму віртуального процесора 8086 («Virtual-8086 Mode Exceptions») пов'язані з тим, що інструкція CLAC не розпізнається у даному режимі.

ВИСНОВКИ ДО РОЗДІЛУ 1

Даний розділ містить теоретичну інформацію про досліджувані сучасні технології Intel, які застосовують для захисту операційних систем шляхом надання захисту виконуваного простору. Такий захист полягає у забороні взаємодії з ненадійними файлами та додатками у певних ділянках пам'яті операційної системи, маючи на меті запобігти будь-якому небажаному впливу на неї. На основі наведених даних можна переконатися, що перелік найбільш «пасуючих» даній меті, на сьогодні навіть стандартизованих, рішень, визначається такими сучасними технологіями, як XDB, PaX, DEP, ASLR, SMEP, SMAP, активно використовуваними на процесорах корпорації Intel. Загалом, вони мають схожий принцип дії – позначення необхідних ділянок пам'яті недоступними для небажаних додатків чи процесів, зводячи їх спроби на отримання доступу до обробки винятків. Це дає змогу захистити операційну систему від багатьох експлойтів, зокрема, тих, що користуються вразливостями типу «переповнення буфера пам'яті», «повернення в бібліотеку», «перевищення привілеїв» і навіть експлойтів механізмів самої обробки винятків. Всі технології є апаратними, однак мають можливість бути програмно емульованими. Реалізація кожної пов'язана з розмежуванням прав доступу, адресацією віртуальної пам'яті, сторінковою організацією пам'яті, що знову-таки виступає на користь їх розгляду та впровадження у ОС з метою її захисту від дій зловмисників.

РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСОБУ

2.1. Аналіз проблеми та постановка завдання

У попередньому розділі було з’ясовано, що кожна досліджувана технологія пов’язана з експлуатацією ядра і застосовується для того, аби запобігти доступу зловмисника до важливих ділянок пам’яті системи з подальшим впливом на ядро. Проте, чи достатньо буде лише однієї певної технології для практичного надання якісного захисту операційної системи? Очевидно, що ні, адже жодна технологія не є досконалою і не може відбивати всі можливі типи атак. Тому відповідно, метою даної роботи є розроблення програмного засобу захисту операційної системи Windows на основі сукупності сучасних технологій Intel, досліджуваних теоретично у першому розділі. Операційна система була обрана із того розрахунку, що у ній, на відміну від інших поширених ОС (Linux, FreeBSD, OpenBSD, NetBSD, XEN), немає можливості програмно керувати сукупністю даних технологій, оскільки для цього необхідно мати привілейований доступ до ядра, що не передбачається Windows. Спираючись на актуальність дослідження, у ньому використано 64-розрядну версію ОС MS Windows 10, оскільки сьогодні вона є найбільш уживаною як у домашньому, так і корпоративному використанні.

Важливо зазначити, що увага в роботі зосереджена не на пошуку і виправленні системних помилок за допомогою програмного засобу, а на запобіганні і стримуванні експлойтів, успішному відбиванні атак. Експлойт – це програма, фрагмент коду або послідовність команд, призначена для здійснення атаки на обчислювальну систему шляхом врахування її вразливостей. Атаки можуть здійснюватися з метою захоплення контролю над системою або спричинення порушення функціонування системи. Виділяють три класи результатів використання вразливостей системи експлойтами:

1) підвищення привілеїв (локальне або віддалене);

2) розкриття інформації;

3) відмова в обслуговуванні.

Усі досліджувані технології мають справу з першим класом, тобто перевищенням привілеїв, що досягається шляхом впровадження експлойтів типу переповнення буфера. Вони можуть впливати на атакований процес на трьох різних рівнях:

1) введення / виконання довільного коду;

2) виконання існуючого коду у вихідному стані програми;

3) виконання існуючого коду у вихідному порядку з довільними даними.

Наприклад, техніка ін'єкції шелл-коду відноситься до (1), а так звана техніка «повернення у бібліотеку» – до (2).

Шелл-код – це частина коду експлойта, що дозволяє після інфікування цільової системи жертви отримати код командної оболонки.

Один із сценаріїв атак полягає в тому, що скориставшись переповненням буфера в програмі, експлойт може записати код до області даних вразливої програми таким чином, що у результаті помилки цей код отримає управління і виконає дії, запрограмовані зловмисником (у більшості випадків це запит виконати програму-оболонку ОС, за допомогою якої зловмисник отримає контроль над системою з правами власника програми, найчастіше – root).

Переповнення буфера також часто виникає, коли розробник програми виділяє деяку область даних (буфер) фіксованої довжини, вважаючи, що цього буде достатньо, але потім, маніпулюючи даними, не перевіряє вихід за її межі. Як результат, вхідні дані займають області пам'яті, їм не призначені, знищивши наявну там інформацію. Дуже часто тимчасові буфери виділяються всередині процедур (підпрограм), пам'ять для яких виділяється в програмному стеку, в якому також зберігаються адреси повернень у підпрограму. Ретельно дослідивши код програми, зловмисник може виявити такий баг, і тепер йому достатньо передати в програму таку послідовність даних, обробивши яку вона помилково замінить адресу повернення в стеку на адресу, потрібну зловмиснику, який та-

кож передав під виглядом даних свій програмний код. Після завершення підпрограми команда повернення з процедури (RET) здійснить передачу управління не викликаній процедурі, а процедурі зловмисника. На цьому контроль над системою отриманий.

Введення коду в адресний простір завдання можливе або шляхом створення виконуваного зіставлення, або шляхом зміни вже існуючого записуваного / виконуваного зіставлення. Першому методу можна запобігти, виконуючи контроль доступу (наприклад, за допомогою досліджуваної технології SMAP). Другому методу можна запобігти, взагалі не дозволяючи створювати доступні для запису / виконувані зіставлення. Хоча це рішення суперечить додаткам, які дійсно потребують таких зіставлень, поки вони не будуть переписані для більш ретельної обробки таких зіставлень, це найкраще можливе рішення. Для цього активно застосовується інструкція NOEXEC досліджуваної технології PaX.

Виконання коду (введеного зловмисником або вже присутнього в адресному просторі завдання) вимагає можливості зміни потоку виконання з використанням вже існуючого коду. Такі зміни відбуваються, коли код розіменовує покажчик на функцію. Зловмисник може втрутитися, якщо такий покажчик зберігається в доступній для запису пам'яті. Оскільки взагалі не мати покажчиків функцій в записуваній пам'яті, на жаль, неможливо (так, як збережені адреси повернення з процедур знаходяться в стеку), тому необхідний інший підхід, який пропонують SMAP і SMEP.

Рандомізація розміщення адресного простору з точки зору захисту від переповнення буфера відноситься до технології, призначеної для того, щоб зробити атаки переповнення на основі стека більш важкими для виконання. Технологія робить це, переставляючи порядок елементів стека. Це означає, що зловмисник не може покладатися на перезапис специфічної для системи інформації шляхом переповнення змінної. Ця форма захисту ефективна проти автоматичних атак, тому що автоматизована програма не може знати, де і як викладена інформація, а для зловмисника ускладнюється маніпулювання процесом, оскільки вимагає більше часу і зусиль.

Варто відзначити, що властивість комбінування захисних механізмів часто неправильно розуміється. Невірним є розгляд окремих частин сукупної системи і на основі цього досягнення висновків щодо її загальної ефективності (або ігнорування розгляду однієї технології, тому що вона недостатньо ефективна без використання іншої, і навпаки). Такий підхід може призвести до помилкових результатів. Розгортання обох видів захисних механізмів захищає від різних експлойтів: там, де одна технологія захисту зазнає невдачі, інша виявиться запобіжною.

Як зазначалося у першому розділі, технології Intel можуть бути програмно емульовані у випадку відсутності їх апаратної підтримки, на що зроблено акцент у даному розділі. У такому разі технології можуть бути впроваджені та активовані для роботи лише за допомогою відповідних системних команд та інструкцій. Проте вони належить до привілейованих, а тому не можуть бути виконані на користувацькому рівні (рівень консольних та графічних програмних додатків та служб). Доступ до привілейованого системного рівня може бути отримано різними шляхами, залежно від операційної системи. У даному випадку, маючи ОС Windows, є можливість виконання привілейованих команд за допомогою механізму програмного драйвера, у вигляді якого й було розроблено засіб захисту.

2.2. Опис засобу та його компонентів

Попередній засіб захисту фактично був програмним драйвером під назвою «SMAP_LOADER_UNLOADER», що містила такі компоненти (див. рис. 2.1):

- папка «fasm»;
- «BUILD.BAT»;

- «install.asm»;
- «readme.txt»;
- «remove.asm»;
- «RUN.BAT»;
- «SMAP_LOADER_UNLOADER.asm»;
- «start.asm»;
- «stop.asm»;
- «Support.cpp».

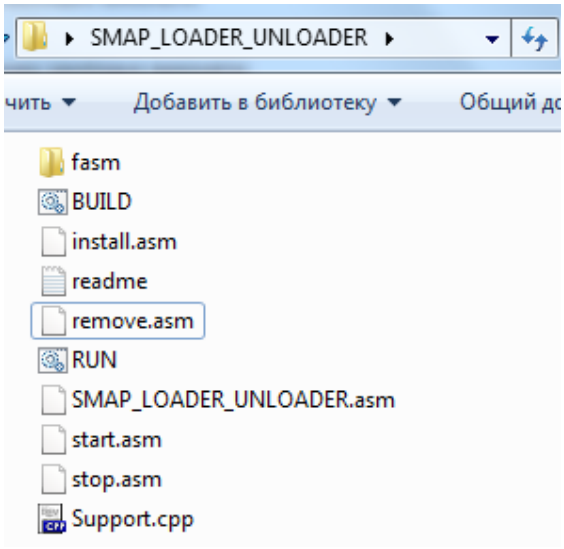


Рис. 2.1 Склад компонентів програмного засобу SMAP_LOADER_UNLOADER

Відповідно, для створення засобу було обрано мови програмування C++ та Assembler.

Файл «Support.cpp» містить програмний код (див. додаток Н) визначення підтримуваних процесором апаратних технологій для усвідомлення користувачем доцільності запровадження захисту засобом. Для отримання необхідної інформації у програмі використовуються макроси інструкції CPUID – __cpuid та __cpuidex. Фрагмент коду компонента програмного засобу:

Support.cpp.cpp

```
class InstructionSet
{
    // клас набору інструкцій процесора
    class InstructionSet_Internal;

public:
    // отримання значень бітів відповідних регістрів процесора, що
    //відповідають за підтримання технологій, а також даних про виробника процесора
    static std::string Vendor(void) { return CPU_Rep.vendor_; }
    static std::string Brand(void)  { return CPU_Rep.brand_; }

    static bool SSE3(void)          { return CPU_Rep.f_1_ECX_[0];  }
    static bool PCLMULQDQ(void)     { return CPU_Rep.f_1_ECX_[1];  }
    static bool MONITOR(void)       { return CPU_Rep.f_1_ECX_[3];  }
    static bool SSSE3(void)         { return CPU_Rep.f_1_ECX_[9];  }
    static bool FMA(void)           { return CPU_Rep.f_1_ECX_[12]; }
    static bool CMPXCHG16B(void)    { return CPU_Rep.f_1_ECX_[13]; }
    static bool SSE41(void)         { return CPU_Rep.f_1_ECX_[19]; }
    static bool SSE42(void)         { return CPU_Rep.f_1_ECX_[20]; }
    static bool MOVBE(void)         { return CPU_Rep.f_1_ECX_[22]; }
    static bool POPCNT(void)        { return CPU_Rep.f_1_ECX_[23]; }
    static bool AES(void)           { return CPU_Rep.f_1_ECX_[25]; }
    static bool XSAVE(void)         { return CPU_Rep.f_1_ECX_[26]; }
    static bool OSXSAVE(void)       { return CPU_Rep.f_1_ECX_[27]; }
    static bool AVX(void)           { return CPU_Rep.f_1_ECX_[28]; }
    static bool F16C(void)          { return CPU_Rep.f_1_ECX_[29]; }
    static bool RDRAND(void)        { return CPU_Rep.f_1_ECX_[30]; }

    static bool MSR(void)           { return CPU_Rep.f_1_EDX_[5];  }
    static bool CX8(void)           { return CPU_Rep.f_1_EDX_[8];  }
    static bool SEP(void)           { return CPU_Rep.f_1_EDX_[11]; }
    static bool CMOV(void)          { return CPU_Rep.f_1_EDX_[15]; }
    static bool CLFSH(void)         { return CPU_Rep.f_1_EDX_[19]; }
    static bool MMX(void)           { return CPU_Rep.f_1_EDX_[23]; }
    static bool FXSR(void)          { return CPU_Rep.f_1_EDX_[24]; }
    static bool SSE(void)           { return CPU_Rep.f_1_EDX_[25]; }
    static bool SSE2(void)          { return CPU_Rep.f_1_EDX_[26]; }

    static bool FSGSBASE(void)      { return CPU_Rep.f_7_EBX_[0];  }
    static bool BMI1(void)          { return CPU_Rep.f_7_EBX_[3];  }
    static bool HLE(void)           { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[4]; }
    static bool AVX2(void)          { return CPU_Rep.f_7_EBX_[5];  }
    static bool SMEP(void)          { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[7]; }
    static bool BMI2(void)          { return CPU_Rep.f_7_EBX_[8];  }
    static bool ERMS(void)          { return CPU_Rep.f_7_EBX_[9];  }
    static bool INVPCID(void)       { return CPU_Rep.f_7_EBX_[10]; }
    static bool RTM(void)           { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[11]; }
    static bool AVX512F(void)       { return CPU_Rep.f_7_EBX_[16]; }
    static bool RDSEED(void)        { return CPU_Rep.f_7_EBX_[18]; }
    static bool ADX(void)           { return CPU_Rep.f_7_EBX_[19]; }
    static bool SMAP(void)          { return CPU_Rep.isIntel_ && CPU_Rep.f_7_EBX_[21]; }
    static bool AVX512PF(void)      { return CPU_Rep.f_7_EBX_[26]; }
    static bool AVX512ER(void)      { return CPU_Rep.f_7_EBX_[27]; }
    static bool AVX512CD(void)      { return CPU_Rep.f_7_EBX_[28]; }
    static bool SHA(void)           { return CPU_Rep.f_7_EBX_[29]; }
```

Рис. 2.2 Фрагмент програмного коду компонента «Support.cpp»

Зазвичай, для використання драйверу потрібен особливий цифровий підпис – свідоцтво про безпеку та надійність даного драйвера від виробника (у даному випадку – самої корпорації Microsoft). Цей підпис-сертифікат долучається до програмного коду додатка та надає змогу використовувати його повсякчас при звичайному користувацькому сценарії вжитку ОС. Проте, у даному випадку, за відсутності такої змоги, відбувається увімкнення спеціальною командою режиму налагодження для розробника драйверів:

bcdedit.exe /set nointegritychecks ON

А також необхідно задіяти меню вибору особливих варіантів для завантаження ОС та обрати пункт, який вимикає необхідну перевірку цифрового підпису. Ця опція працюватиме під час однієї робочої сесії. При аварійному пе-

резавантаженні, чи простому вимкнені наступна сесія вже потребуватиме обов'язковий цифровий підпис.

Так, як драйвер Windows – це частина ядра, то виконання такого додатку досить сильно відрізняється від звичайного. Для виконання програмного коду драйвера необхідні інші програмні додатки, що надають змогу «звертатися» до нього, оскільки драйвер працюватиме у якості частини ОС, тобто на більш привілейованому рівні, ніж класична служба. Драйвер має чіткі програмні стани, що надають йому можливість реагувати на зовнішні події (сигнали), спричинені іншими додатками. Так, наприклад, драйвер може отримати сигнал на виконання, але для успішного опрацювання такого сигналу необхідні відповідні процедури.

У драйвері містяться процедури для обслуговування подій виконання, припинення, саме тому такий програмний засіб цілком підходить для демонстрації виконання привілейованих команд. Самі ж процедури містяться у загальній структурі, яку обслуговує основна процедура драйвера – головна ділянка, яка виконує усі необхідні етапи для подальшого функціонування.

Драйвер, на відміну від класичного засобу, не має чітких механізмів для сповіщення результатів своєї роботи та опрацювання невідкладних ситуацій. А тому було задіяно наступний принцип: коли драйвер завантажується, то він виконує необхідні команди для активації технологій та використовує вбудований системний гучномовець. Те ж саме відбувається і при деактивації. Цей саме принцип використовується і в удосконаленому програмному засобі.

На чолі всіх запрограмованих процесів стоять bat-сценарії, що виконують побудову застосунків та в подальшому виконують їх. Тому для коректної роботи драйвера необхідно скопіювати його файл до системної директорії, звідки можна його виконувати. Це необхідно для того, щоб уникнути проблем сумісності та розташування виконуваного файлу. Директорія «system32\drivers» є директорією за замовчуванням для подібних сценаріїв і доки файл розміщено там, його неможливо фізично видалити. Усю роботу з виконання сценарію тестування драйвера здійснює BAT-файл «RUN.BAT».

Для більшої зручності та точності у обох випадках розроблення програмного засобу було зроблено саме комплекс застосунків, створених на мові Assembler. Це дозволило уникнути використання громіздкого програмного середовища для створення драйверів, а також надало змогу керувати змістом програмного засобу. Дана мова програмування надає можливість не тільки безпосередньо виконувати різноманітні інструкції процесора, а й дослідити і скеровувати процес створення програмного додатку. Такий контроль вирішив одну з головних проблем розробки драйверів під операційну систему Windows 10 – зміну дизайну програмного середовища для функціонування коду на рівні драйвера. А саме, сегмент релокацій. З огляду на безпеку, він необхідний навіть при розробці 64-бітного засобу. Але при його класичному змісті, виконання такого драйвера буде блокуватися операційною системою. Тут вийшло уникнути даної проблеми, використовуючи асемблер «fasm» (від англ.«Flat Assembler»).

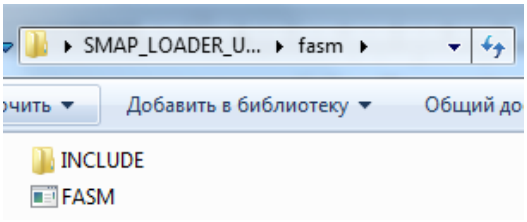


Рис. 2.3 Вміст папки «fasm»

Підпапка «INCLUDE» містить файли програмних включень у вигляді службових файлів, що необхідні для розробки та асемблювання додатків мовою асемблера на платформі win32 (див. рис. 2.4).

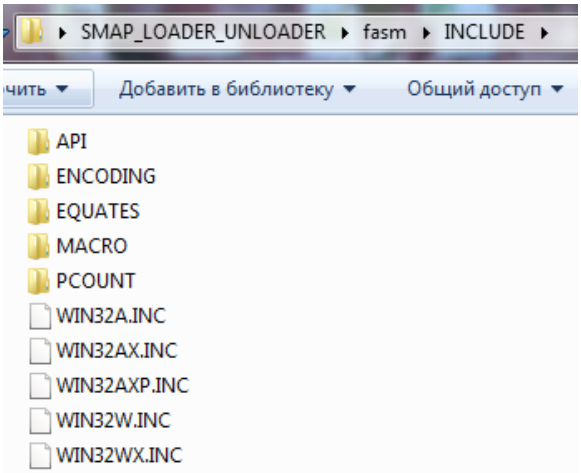


Рис. 2.4 Вміст підпапки «INCLUDE» папки «fasm»

Так, підпапка «API» містить службові файли, необхідні для звернень до функцій бібліотек (див. рис. 2.5), «ENCODING» - Юнікод-файли (див. рис. 2.6), «EQUATES» - стандартні константи і структури, використовувані функціями ОС (див. рис. 2.7), «MACRO» - необхідні макроси (див. рис. 2.8), «PCOUNT» - файли з кількістю аргументів для виклику процедур основних файлів-компонентів драйвера (див. рис. 2.9).

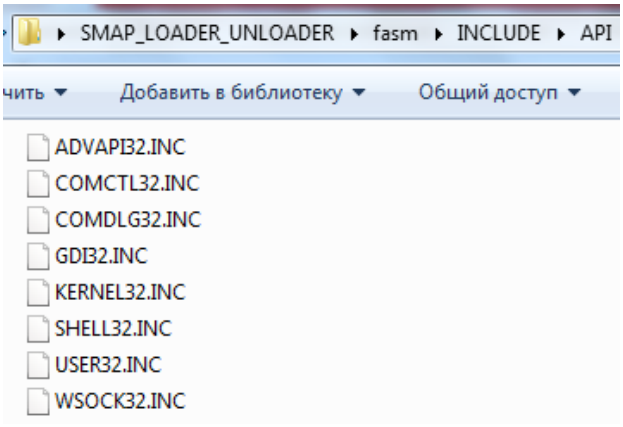


Рис. 2.5 Вміст підпапки «API» папки «INCLUDE»

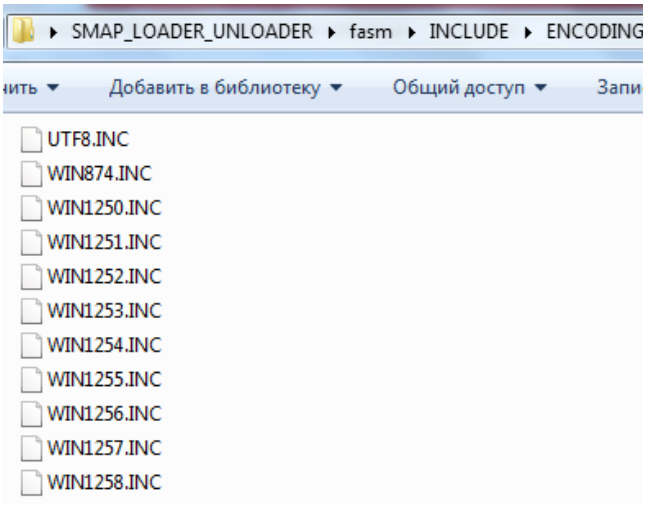


Рис. 2.6 Вміст підпапки «ENCODING» папки «INCLUDE»

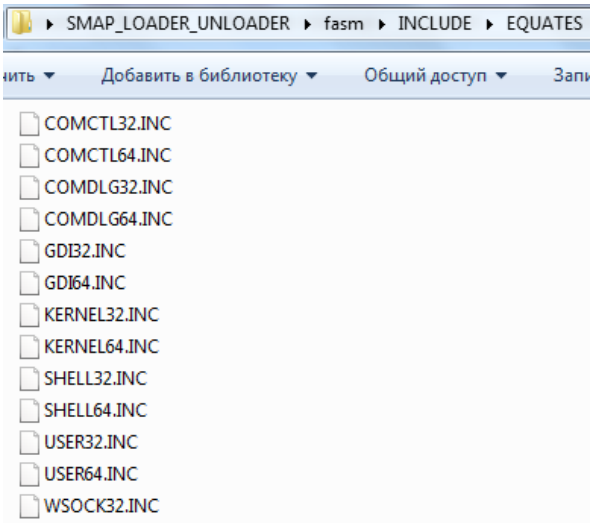


Рис. 2.7 Вміст підпапки «EQUATES» папки «INCLUDE»

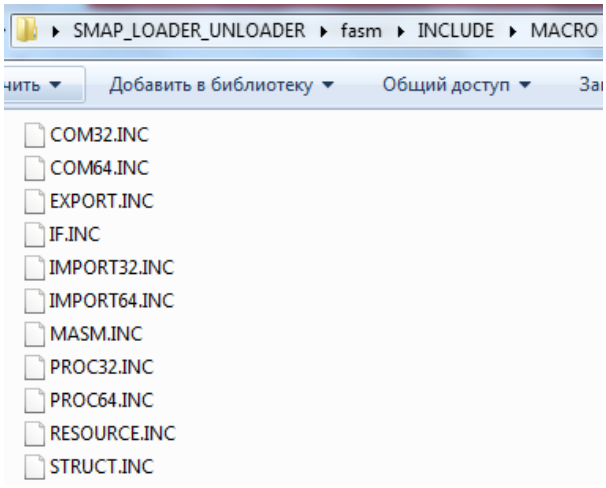


Рис. 2.8 Вміст підпапки «MACRO» папки «INCLUDE»

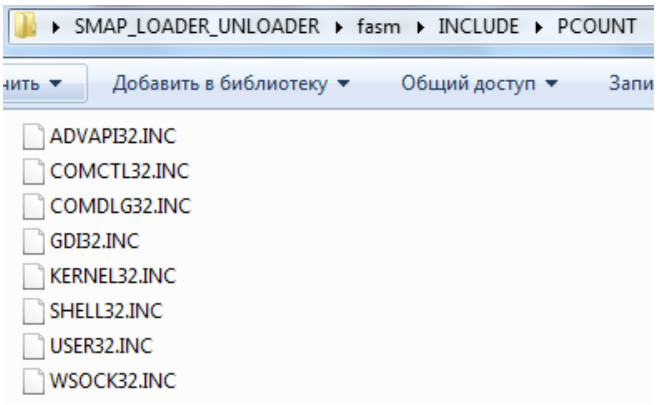


Рис. 2.9 Вміст підпапки «PCOUNT» папки «INCLUDE»

Основна робота програмного драйвера розпочинається виконанням застосунку «install.asm» (див. додаток К). У ньому відбуваються наступні операції:

1. Початкове читання отриманого з командного рядку шляху розміщення драйвера. Цей етап можна описати як початковий парсинг параметрів командного рядку.

2. Збереження адреси отриманого рядка з вказівником на фрагмент з ім’ям драйвера для системного виклику в майбутньому.

3. Отримання дескриптора бази даних диспетчеру скеровування системними службами за допомогою системного виклику OpenSCManagerA (див. рис. 2.10) отримує дескриптор. Також одразу виконується перевірка на виконання цього запиту.

4. За допомогою отриманого дескриптора системних сервісів застосунок може створити власний екземпляр та додати його до бази даних. Цю роботу виконує системний виклик CreateServiceA (див. рис. 2.11). Результуюче значення – це дескриптор власної служби, тобто майбутній вказівник на розташування драйвера в пулі процесів операційної системи.

Фрагменти програмного коду застосунку:

```
75
76  scan_name:
77          dec rdi
78          cmp byte [rdi-1],'\'
79          jnz scan_name
80          mov [driver_name_address],rdi
81
82          mov r8d,    SC_MANAGER_CONNECT\
83                  or SC_MANAGER_CREATE_SERVICE\
84                  or SC_MANAGER_ENUMERATE_SERVICE\
85                  or SC_MANAGER_LOCK\
86                  or SC_MANAGER_QUERY_LOCK_STATUS\
87                  or SC_MANAGER_MODIFY_BOOT_CONFIG\
88                  or STANDARD_RIGHTS_REQUIRED
89          xor edx,edx
90          xor ecx,ecx
91          call    qword [OpenSCManagerA]
```

Рис. 2.10 Фрагмент програмного коду компонента «install.asm»

```
104  L0:     xchg    rbx,rax
105          mov qword [rsp+8*(4+8)],0
106          mov qword [rsp+8*(4+7)],0
107          mov qword [rsp+8*(4+6)],0
108          mov qword [rsp+8*(4+5)],0
109          mov qword [rsp+8*(4+4)],0
110          lea rax,[driver_path_and_name]
111          mov qword [rsp+8*(4+3)],rax
112          mov qword [rsp+8*(4+2)],SERVICE_ERROR_NORMAL
113          mov qword [rsp+8*(4+1)],SERVICE_DEMAND_START
114          mov qword [rsp+8*(4+0)],SERVICE_KERNEL_DRIVER
115          mov r9d,    SERVICE_QUERY_CONFIG\
116                  or SERVICE_CHANGE_CONFIG\
117                  or SERVICE_QUERY_STATUS\
118                  or SERVICE_ENUMERATE_DEPENDENTS\
119                  or SERVICE_START\
120                  or SERVICE_STOP\
121                  or SERVICE_PAUSE_CONTINUE\
122                  or SERVICE_INTERROGATE\
123                  or SERVICE_USER_DEFINED_CONTROL\
124                  or STANDARD_RIGHTS_REQUIRED
125          mov r8,[driver_name_address]
126          mov rdx,r8
127          mov rcx,rbx
128          call    qword [CreateServiceA]
129          or  rax,rax
130          jz  L1
131
132          mov rcx,rbx
133          call    qword [CloseServiceHandle]
134
135  egzyd:  xor ecx,ecx
136          call    qword [ExitProcess]
137
138  L1:     mov r9d, MB_OK or MB_ICONERROR
```

Рис. 2.11 Фрагмент програмного коду компонента «install.asm»

У результаті отримано завантажений драйвер, готовий для подальшого використання. Наступний етап – це виконання застосунку «start.asm» (див. додаток Л). Перші його кроки нагадують попередній застосунок задля самостійності кожного з додатків. За допомогою даного компоненту відбувається:

1. Початковий парсинг і приєднання до існуючої бази даних системних служб.

2. Отримання вказівника на драйвер за допомогою системного виклику OpenServiceA.

3. Звернення до драйвера шляхом надсилання сигналу на виконання за допомогою системної процедури StartServiceA.

Фрагмент програмного коду даного застосунку:

```
 95  L0:     mov r8d,    SERVICE_QUERY_CONFIG\
 96                  or SERVICE_CHANGE_CONFIG\
 97                  or SERVICE_QUERY_STATUS\
 98                  or SERVICE_ENUMERATE_DEPENDENTS\
 99                  or SERVICE_START\
100                  or SERVICE_STOP\
101                  or SERVICE_PAUSE_CONTINUE\
102                  or SERVICE_INTERROGATE\
103                  or SERVICE_USER_DEFINED_CONTROL\
104                  or STANDARD_RIGHTS_REQUIRED
105         lea rdx,[driver_name]
106         xchg    rcx,rax
107         call    qword [OpenServiceA]
108         or  rax,rax
109         jnz L1
110
111         mov r9d,MB_OK or MB_ICONERROR
112         lea r8,[ErrMsgCaption3]
113         lea rdx,[ErrMsgText3]
114         xor ecx,ecx
115         call    qword [MessageBoxA]
116
117         mov ecx,3
118         call    qword [ExitProcess]
119
120 L1:     xchg    rbx,rax
121         xor r8,r8
122         xor edx,edx
123         mov rcx,rbx
124         call    qword [StartServiceA]
125         or  rax,rax
126         jnz L2
127
128         mov r9d,MB_OK or MB_ICONERROR
129         lea r8,[ErrMsgCaption3]
130         lea rdx,[ErrMsgText4]
131         xor ecx,ecx
132         call    qword [MessageBoxA]
133
134         mov ecx,4
135         call    qword [ExitProcess]
136
137 L2:     mov rcx,rbx
138         call    qword [CloseServiceHandle]
```

Рис. 2.12 Фрагмент програмного коду компонента «start.asm»

Далі виконання передається основному файлу програмного засобу – «SMAP_LOADER_UNLOADER.asm» (див. додаток Л). Спершу відбувається виконання процедури DriverEntry: здійснюється початкова ініціалізація тексту в системному Юнікод-форматі, який використовують механізми ядра ОС (див. рис. 2.13). Цей текст міститься у змінних pNT_DEVICE_NAME та pDOS_DEVICE_NAME. Потім ініційовані текстові змінні передаються до IoCreateDevice – ця процедура створює об’єкт-драйвер – змінну, що надасть змогу звертатися до драйвера за посиланням.

Потім відбувається створення посилання за допомогою IoCreateSymbolicLink, а далі – завантаження до основної структури об'єкту-драйвера посилання на всі процедури, що будуть використані (див. рис. 2.13).

```
; Процедура-опрацьовувач драйвера. Містить точку входу та виконує
; первинну ініціалізацію
proc DriverEntry pDriverObject:QWORD, pusRegistryPath:QWORD
    local usDeviceName:UNICODE_STRING, usSymbolicLinkName:UNICODE_STRING
    local pDeviceObject:QWORD

    Mov [pDriverObject], Rcx
    mov [pusRegistryPath], rdx

    invoke RtlInitUnicodeString, addr usDeviceName, addr pNT_DEVICE_NAME
    invoke RtlInitUnicodeString, addr usSymbolicLinkName, addr pDOS_DEVICE_NAME

    invoke IoCreateDevice, [pDriverObject], sizeof.DEVICE_EXTENSION, addr usDeviceName, \
                           FILE_DEVICE_UNKNOWN, 0, TRUE, addr pDeviceObject

    cmp rax, STATUS_SUCCESS
    je @f
    ; Гучномовець подає сигнал про помилку
    ; Сигнал невдалої спроби створення програмного пристрою
    fastcall MakeBeep, 500
    mov rax, STATUS_DEVICE_CONFIGURATION_ERROR
    jmp L1

@@: invoke IoCreateSymbolicLink, addr usSymbolicLinkName, addr usDeviceName
    cmp rax, STATUS_SUCCESS
    je @f

    ; Гучномовець подає сигнал про помилку
    ; Сигнал невдалої спроби створення символьного посилання
    invoke IoDeleteDevice, [pDriverObject]
    fastcall MakeBeep, 400
    mov rax, STATUS_DEVICE_CONFIGURATION_ERROR
    jmp L1
; Після успіху попередніх процедур, розміщуються дані для подальшої роботи драйвера
@@: mov rax, [pDriverObject]
    lea rcx, [DriverUnload]

    ; Розміщення вказівника на процедуру-опрацьовувач вилучення драйвера
    ; у системній структурі
    mov [rax + DRIVER_OBJECT.DriverUnload], rcx
    lea rcx, [DriverCreate]

    ; Розміщується вказівник на процедуру-опрацьовувач для створення
    ; програмного драйвера у системній структурі
    mov [rax + DRIVER_OBJECT.MajorFunction + IRP_MJ_CREATE * 8], rcx
    lea rcx, [DriverRead]
```

Рис. 2.13 Фрагмент програмного коду компонента «SMAP_LOADER_UNLOADER.asm»

Саме у цьому застосунку відбувається процедура DriverCreate DriverUnload, яка включає в себе виконання інструкції stac, передбачена саме для звершення найважливішого етапу у роботі драйвера – для активації досліджуваної технології SMAP шляхом перемикання визначного біта керуючого регістра. Вслід за нею визначено процес виконання шкідливого коду. Це зроблено для того, щоб продемонструвати захисну дію самої технології (а отже, і драйвера) на практиці та прослідкувати, до чого це призведе. Шкідливий код містить у своїй структурі звернення до адрес простору памяті користувача, від чого у режимі супервізора SMAP має захищати.

Наступною визначено процедуру DriverUnload з інструкцією clac, за допомогою якої є можливою деактивація технології (див. рис. 2.14).

```
align 16
proc DriverCreate pDeviceObject:QWORD, ppIRP:QWORD
    local Status:QWORD, Info:QWORD
    mov [pDeviceObject], rcx
    mov [ppIRP], rdx
    ; Виконується інструкція нульового рівня для задіення технології SMAP
    ; Про вдалість даної операції сповістить сигнал системного гучномовця
    ;--------------------------
    stac
    fastcall MakeBeep, 1000
    ;--------------------------

    ; Для демонстрації роботи технології здійснюється виконання небезпечного коду
    mov rax,    [QWORD 0x400000]

    mov [Status], STATUS_SUCCESS
    mov [Info], 0H
    mov rdx, [ppIRP]
    mov rax, [Status]
    mov [rdx + 30H], rax
    mov rax, [Info]
    mov [rdx + 38H], rax
    invoke IoCompleteRequest, [ppIRP], IO_NO_INCREMENT
    mov rax, [Status]
    ret
endp
align 16
proc DriverUnload pDriverObject:QWORD
    local usSymbolicLinkName:UNICODE_STRING

    mov [pDriverObject], rcx

    invoke RtlInitUnicodeString, addr usSymbolicLinkName, addr pDOS_DEVICE_NAME
    invoke IoDeleteSymbolicLink, addr usSymbolicLinkName

    ; Виконується інструкція нульового рівня для зупинення дії технології SMAP
    ; Про вдалість даної операції сповістить сигнал системного гучномовця
    ;--------------------------
    clac
    fastcall MakeBeep, 800
    ;--------------------------
    mov rax, [pDriverObject]
    invoke IoDeleteDevice, [Rax + DRIVER_OBJECT.DeviceObject]
    ret
endp
```

Рис. 2.14 Фрагмент програмного коду застосунку «SMAP_LOADER_UNLOADER.asm»

Програмний драйвер також містить активацію і деактивацію ASLR за допомогою попередньо визначених процедур (див. рис. 2.15, 2.16).

```
__enable_Aslr proc PE:LOADED_IMAGE
    invoke __is_Dynamic_Base2, PE
    .if rax != NULL
        ;ASLR is already Enabled
        jmp Exit
    .else
    mov rbx, PE.FileHeader
    or dword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics, IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
    .endif
    ;ASLR Enabled
    ret
__enable_Aslr EndP
```

Рис. 2.15 Фрагмент програмного коду застосунку «ASLR.asm»

```
__disable_Aslr proc PE:LOADED_IMAGE
    invoke __is_Dynamic_Base2, PE
    .if !eax
        ;ASLR is already Disabled
        jmp Exit
    .else
    xor rdx, rdx
    xor rax, rax
    mov rbx, PE.FileHeader
    mov rax, IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
    not rax
    and dword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics, rax
    .endif
    ;ASLR Disabled
    ret
__disable_Aslr EndP
```

Рис. 2.16 Фрагмент програмного коду застосунку «ASLR.asm»

```
__is_Dynamic_Base2 proc PE:LOADED_IMAGE
    mov rdi, PE.FileHeader
    mov rdi, qword ptr [rdi].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics
    and rdi, IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
    .if rdi == IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
        mov rax, TRUE
        ret
    .else
        mov rax, FALSE
        ret
    .endif
__is_Dynamic_Base2 EndP
```

Рис. 2.17 Фрагмент програмного коду застосунку «ASLR.asm»

Компонент драйвера «DEP.asm» містить, відповідно, програмну активацію та деактивацію технології DEP, що можна побачити на рис. 2.18 та рис. 2.19 відповідно.

```
__enable_DEP proc PE:LOADED_IMAGE
    invoke isDEP, PE
    .if eax != NULL
        ;DEP is already Enabled
        jmp Exit
    .else
    mov rbx, PE.FileHeader
    or dword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics, IMAGE_DLLCHARACTERISTICS_NX_COMPAT
    .endif
    ;DEP Enabled
    ret
__enable_DEP EndP
```

Рис. 2.18 Фрагмент програмного коду застосунку «DEP.asm»

```
__disable_DEP proc PE:LOADED_IMAGE
    invoke isDEP, PE
    .if eax == NULL
        ;DEP is already Disabled
        jmp Exit
    .else
    xor rdx, rdx
    xor rax, rax
    mov rbx, PE.FileHeader
    mov rax, IMAGE_DLLCHARACTERISTICS_NX_COMPAT
    not rax
    and dword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics, rax
    .endif
    ;DEP Disabled
    ret
__disable_DEP EndP
```

Рис. 2.19 Фрагмент програмного коду застосунку «DEP.asm»

```
_is_DEP proc PE:LOADED_IMAGE
   mov rbx, PE.FileHeader
   mov rbx, qword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics
   and rbx, IMAGE_DLLCHARACTERISTICS_NX_COMPAT
   .if rbx == IMAGE_DLLCHARACTERISTICS_NX_COMPAT
       mov rax, TRUE
       ret
   .else
       mov rax, FALSE
       ret
   .endif
_is_DEP EndP
```

Рис. 2.20 Фрагмент програмного коду застосунку «DEP.asm»

Окрім того, програмний код здійснює активацію технології PaX викликом процедури enable_PaX:

```
__enable_Pax proc PE:LOADED_IMAGE
    invoke __is_CFG, PE
    .if rax != NULL
        ;Pax is already Enabled
        jmp Exit
    .else
    mov rbx, PE.FileHeader
    or dword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics, IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
    .endif
    ;Pax Enabled
    ret
__enable_Pax EndP
```

Рис. 2.21 Фрагмент програмного коду застосунку «PaX.asm»

І її деактивацію на вибір користувача викликом процедури disable_PaX:

```
__disable_Pax proc PE:LOADED_IMAGE
    invoke __is_CFG, PE
    .if !eax
        ;Pax is already Disabled
        jmp Exit
    .else
    xor rdx, rdx
    xor rax, rax
    mov rbx, PE.FileHeader
    mov rax, IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
    not rax
    and dword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics, rax
    .endif
    ;Pax Disabled
    ret
__disable_Pax EndP
```

Рис. 2.22 Фрагмент програмного коду застосунку «PaX.asm»

```
__is_CFG proc PE:LOADED_IMAGE
    mov rbx, PE.FileHeader
    mov rbx, dword ptr [rbx].IMAGE_NT_HEADERS64.OptionalHeader.DllCharacteristics
    and rbx, 4000h
    .if rbx == 4000h
        mov rax, TRUE
        ret
    .else
        mov rax, FALSE
        ret
    .endif
    mov ebx, PE.FileHeader

__is_CFG EndP
```

Рис. 2.23 Фрагмент програмного коду застосунку «PaX.asm»

Оскільки даним застосунком також передбачено виконання тестування впроваджених технологій на протидію атакам, то до переліку керуючих застосунків також входить виконуваний файл «tests.asm».

Останнім керуючим застосунком є файл «stop.asm» (див. додаток М). Завдяки йому з'являється можливість для зупинки дії технології. Його можливості та процес виконання:

1. На початку також виконуються службові дії. Аналізується ім'я додатку.

2. Відбувається приєднання до існуючої бази системних служб.

3. Отримання вказівника на драйвер.

4. Коли були отримані всі необхідні вказівники, виконується аналіз поточного стану служби драйверу. Цю роботу виконує QueryServiceStatus (див. рис. 2.19). У результаті буде отримано поточний стан драйвера, який згідно засобу мусить бути SERVICE_ACTIVE.

5. Наступним кроком є зупинка роботи драйвера. Після успішного виконання виклику ControlService (див. рис. 2.24) із запитом на зупинку SERVICE_CONTROL_STOP можна переходити до виконання останнього застосунку.

```
126
127 L1:     xchg    rbx,rax
128        lea rdx,[Service_status]
129        mov rcx,rbx
130        call    qword [QueryServiceStatus]
131        cmp byte [Service_status + 4],SERVICE_STOPPED
132        jz  ess
133
134        lea r8,[Service_status]
135        mov edx,SERVICE_CONTROL_STOP
136        mov rcx,rbx
137        call    qword [ControlService]
138        or  rax,rax
139        jnz L2
140
141 ess:    mov r9d,MB_OK or MB_ICONERROR
142        lea r8,[ErrMsgCaption3]
143        lea rdx,[ErrMsgText4]
144        xor ecx,ecx
145        call    qword [MessageBoxA]
```

Рис. 2.24 Фрагмент програмного коду застосунку «stop.asm»

Файл-компонент «remove.asm» (див. додаток М) за необхідності видаляє драйвер, який завантажено у фоновому режимі, зі списку активних системних служб.

1. Отримується вказівник на базу даних системних служб, після чого отримується вказівник саме на драйвер.

2. CloseServiceHandle – цей системний виклик звільняє драйвер і можна при роботі скрипта видалити виконуваний файл із системної директорії (див. рис. 2.25). Після успішного виконання цього виклику, виконуються всі необхідні перевірки і закриваються всі системні дескриптори. Це надає можливість безпечно виконати зупинку виконання засобу.

```
141  ess:    mov r9d,MB_OK or MB_ICONERROR
142          lea r8,[ErrMsgCaption3]
143          lea rdx,[ErrMsgText4]
144          xor ecx,ecx
145          call    qword [MessageBoxA]
146
147          mov ecx,4
148          call    qword [ExitProcess]
149
150  L2:     mov rcx,rbx
151          call    qword [CloseServiceHandle]
152
153  egzyd:  xor ecx,ecx
154          call    qword [ExitProcess]
155
156  t00_10ng:
157          mov r9d,MB_OK or MB_ICONERROR
158          lea r8,[ErrMsgCaption1]
159          lea rdx,[ErrMsgText1]
160          xor ecx,ecx
161          call    qword [MessageBoxA]
162
163          mov ecx,1
164          call    qword [ExitProcess]
165
166  shell_no_drv:
167          mov r9d,MB_OK or MB_ICONERROR
168          lea r8,[ErrMsgCaption1]
169          lea rdx,[ErrMsgText1_nodrv]
170          xor ecx,ecx
171          call    qword [MessageBoxA]
```

Рис. 2.25 Фрагмент програмного коду застосунку «remove.asm»

Таким чином, розроблений програмний драйвер містить всі необхідні функції програмного засобу, що відповідає за програмну підтримку визначених технологій мікропроцесора і надає користувачеві можливість працювати з ними у системі.

2.3. Демонстрація практичного застосування засобу

Спершу, перед виконанням практичної реалізації основної частини програмного засобу – драйвера, бажано здійснити перевірку процесора на підтримку основних необхідних технологій, що здійснюється шляхом виконання компоненту засобу «Support.cpp». Проте, у разі впевненості користувача у підтримці технології на його ПК, даний крок не є обов'язковим. Для ознайомлення користувача з можливостями процесора у підтримці інших не менш важливих технологій та функцій їх перевірку також було залишено у складі програмного коду.

Реалізація даної частини програмного засобу:

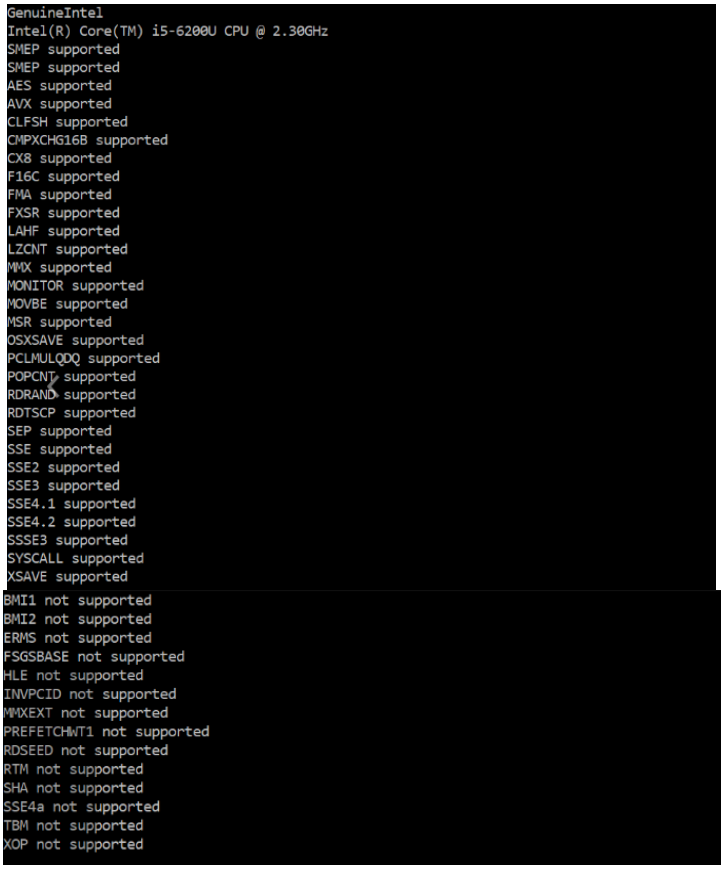


Рис. 2.26 Демонстрація реалізації програмного коду

Задля успішного застосування основної частини розробленого засобу (тобто саме програмного драйвера) на практиці та перевірки його роботи, необхідно від імені адміністратора виконати спершу файл-компонент «BUILD.BAT»

(див. додаток П). Це необхідно для передачі файлів з джерельним кодом програмному асемблеру «fasm», у результаті чого буде отримано програмні застосунки засобу, які є важливою складовою для виконання скрипта «RUN.BAT». Результат виконання скрипта «BUILD.BAT» залежить від команди, яку введе користувач. Перелік команд міститься у файлі «readme.txt». Його вміст (див. рис. 2.27, додаток Р):

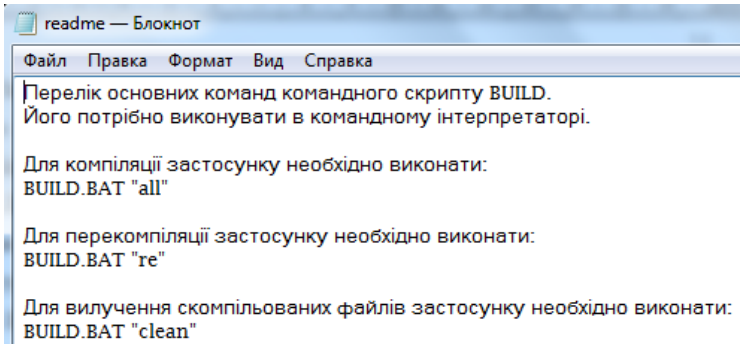


Рис. 2.27 Вміст файлу «readme.txt»

Результати виконання команд наступні (див. рис. 2.28-2.29).

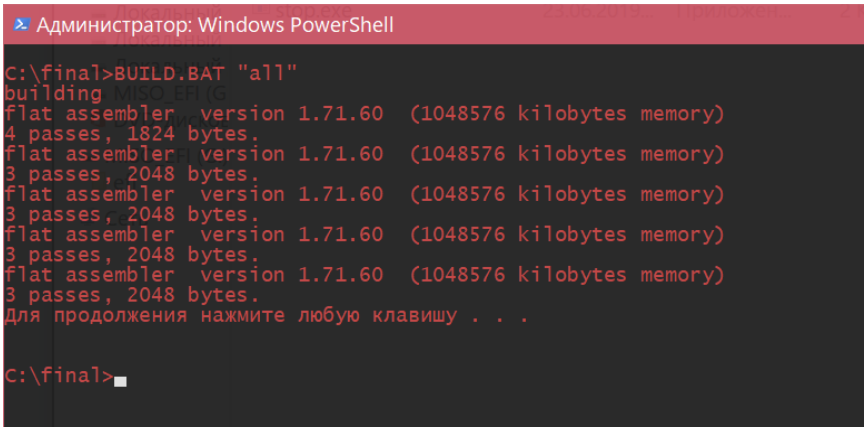


Рис. 2.28 Результат виконання команди «BUILD.BAT "all"»

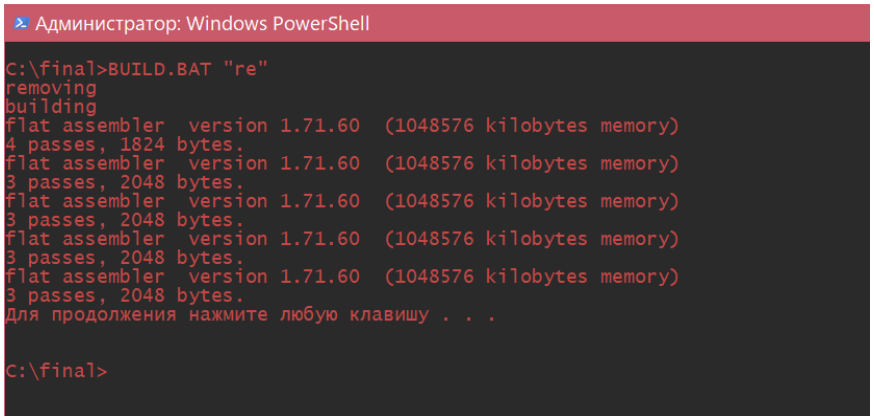


Рис. 2.29 Результат виконання команди «BUILD.BAT " re"»

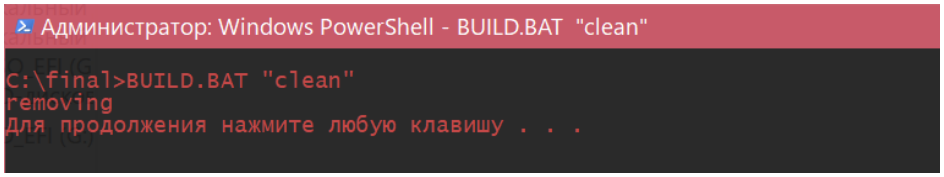


Рис. 2.30 Результат виконання команди «BUILD.BAT "clean"»

Скрипт «RUN.BAT» виконує всі необхідні етапи для завантаження та роботи драйвера (див. додаток С).

Запустивши удосконалений програмний додаток до виконання, можна побачити основне вікно виконання розробленої програми. Так, вона містить привітання, меню, що складається з основних чотирьох компонентів: Файл, Налаштування, Тести, Довідка. Також доступний перехід на конкретні процедури – Налаштування чи Тести. Окрім того, міститься кнопка виходу з програми (див. рис. 2.31).

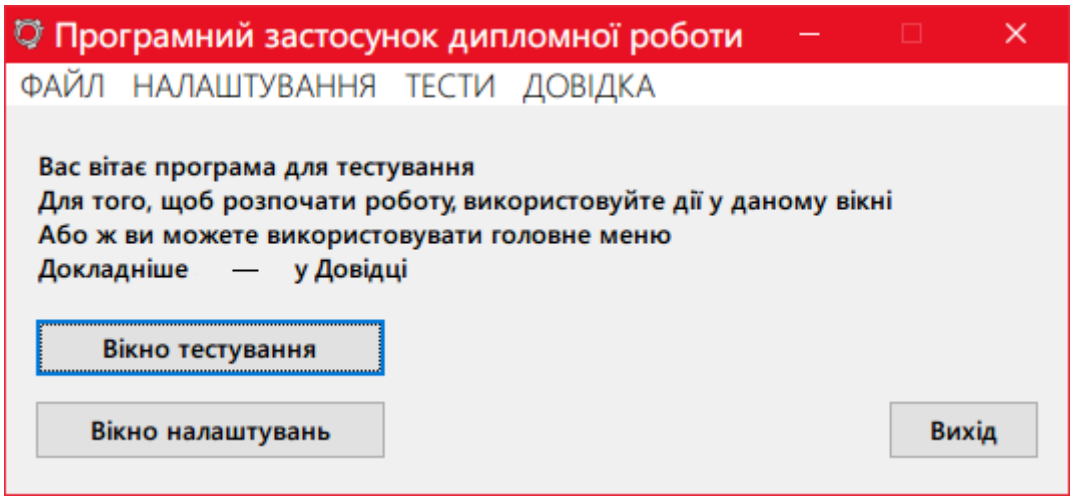


Рис. 2.31 Основне вікно розробленого додатку

Так, з переходом до налаштувань програми користувачеві надається можливість налаштувати активацію необхідних йому технологій (див. рис. 2.32 - 2.36).

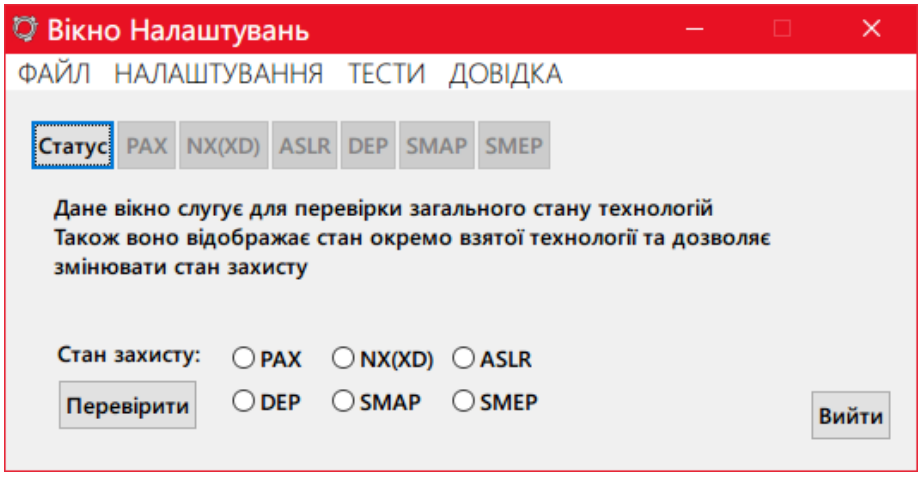


Рис. 2.32 Вікно налаштувань розробленого додатку

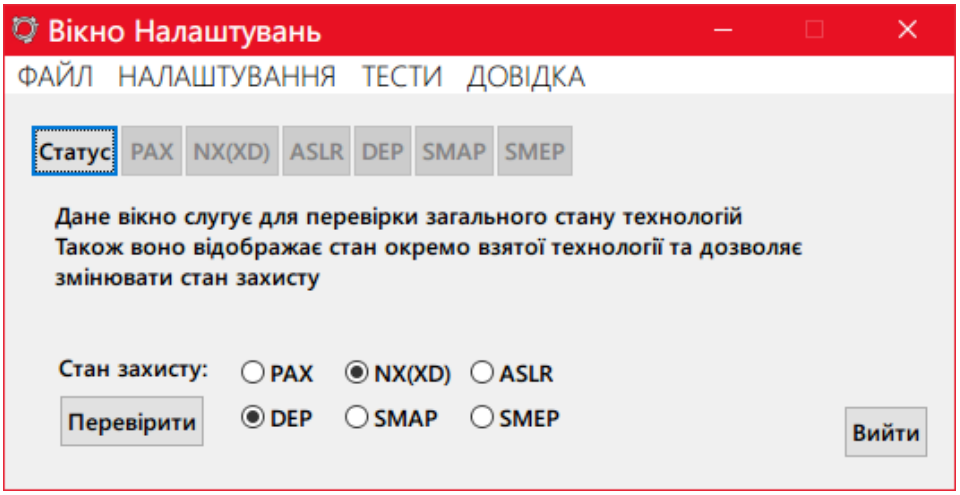


Рис. 2.33 Вікно налаштувань розробленого додатку з варіантом вибору технологій до активації

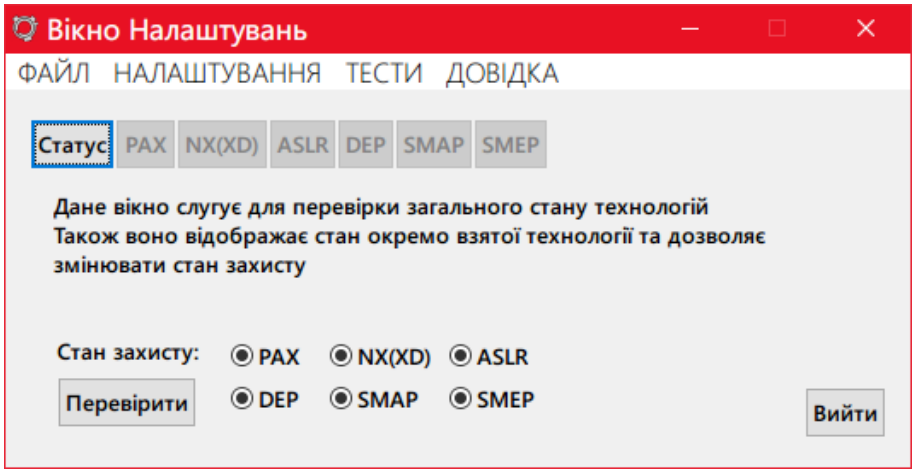


Рис. 2.34 Вікно налаштувань розробленого додатку з варіантом вибору технологій до активації

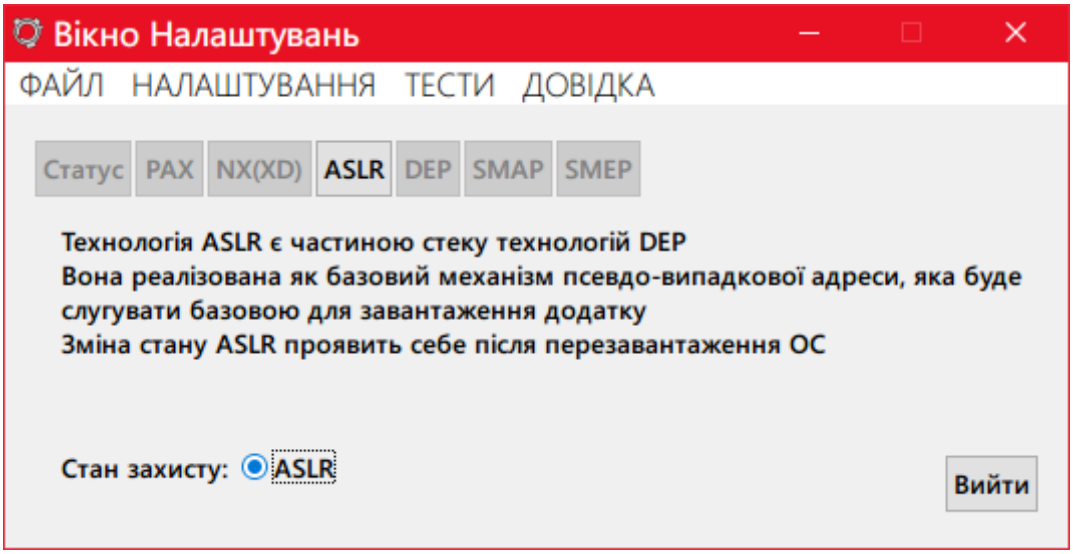


Рис. 2.35 Вікно налаштувань розробленого додатку з перевіркою стану захисту системи за допомогою окремої технології

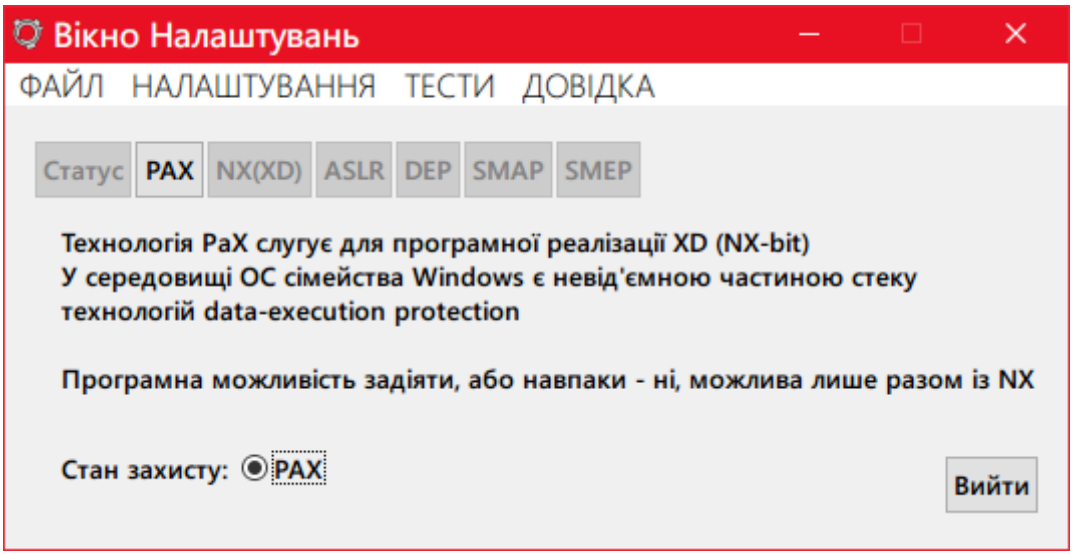


Рис. 2.36 Вікно налаштувань розробленого додатку з перевіркою стану захисту системи за допомогою окремої технології

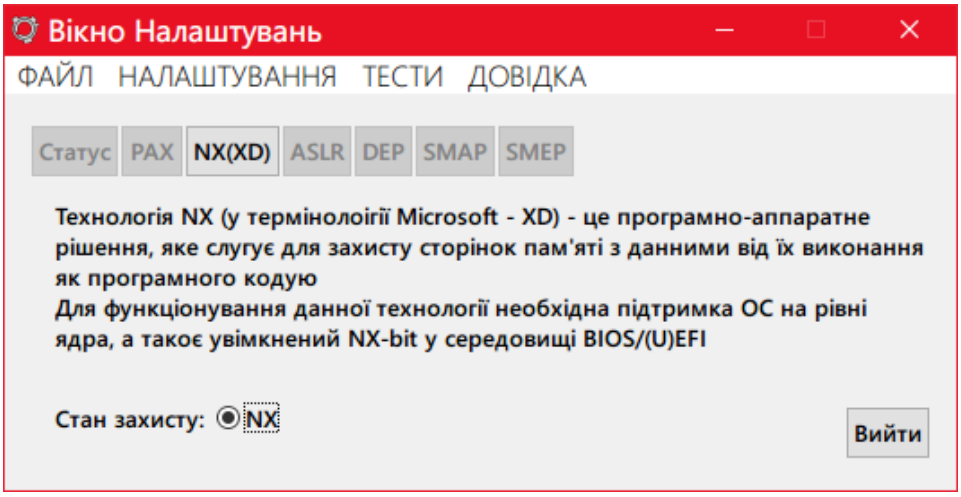


Рис. 2.37 Вікно налаштувань розробленого додатку з перевіркою стану захисту системи за допомогою окремої технології

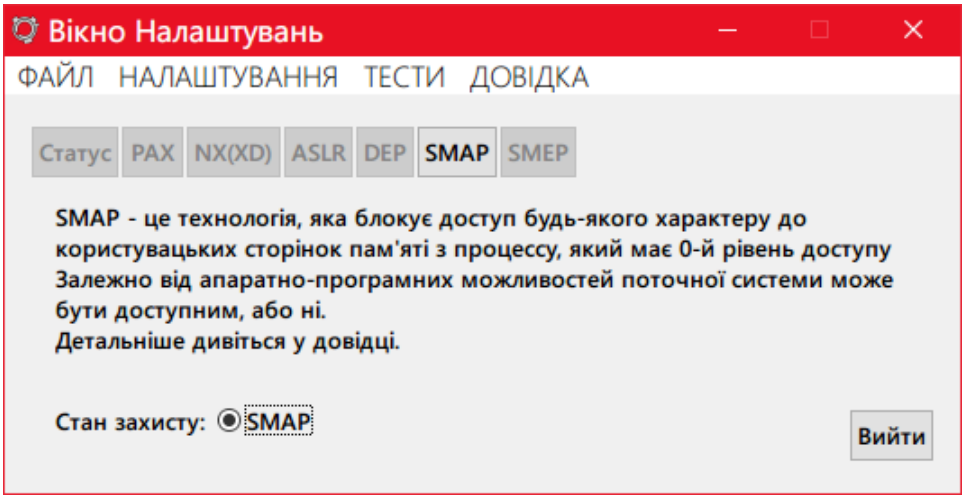


Рис. 2.38 Вікно налаштувань розробленого додатку з перевіркою стану захисту системи за допомогою окремої технології

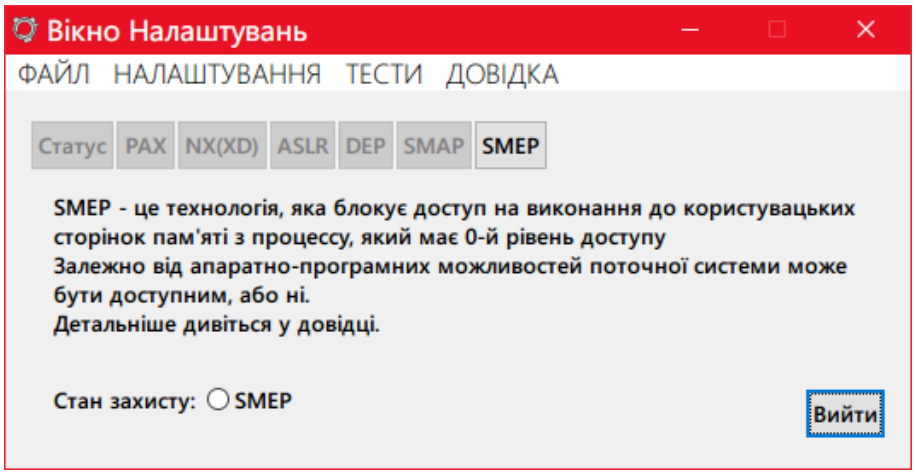


Рис. 2.39 Вікно налаштувань розробленого додатку з перевіркою стану захисту системи за допомогою окремої технології

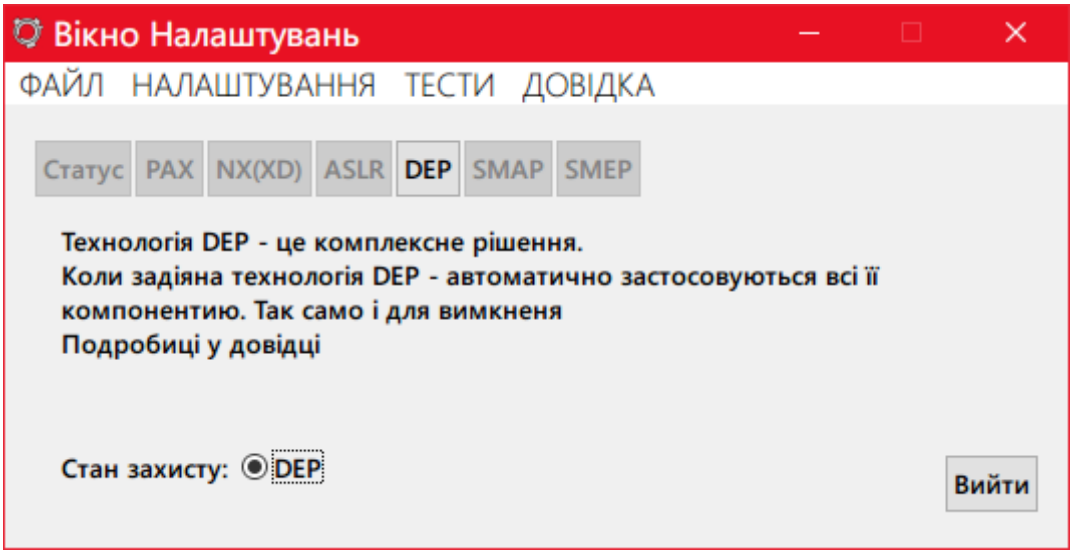


Рис. 2.40 Вікно налаштувань розробленого додатку з перевіркою стану захисту системи за допомогою окремої технології

Як вже зазначалося, у момент активації технології драйвером, користувач чутиме характерний звуковий сигнал гучномовця. Так само і у момент її деактивації, а також у момент генерації помилки. Задля розрізнення цих процесів на слух, використано наступне розмежування: для вдалих операцій лунає високочастотний звуковий сигнал, а для помилок – низькочастотний.

При переході до пункту меню Тести, користувач бачитиме наступне вікно:

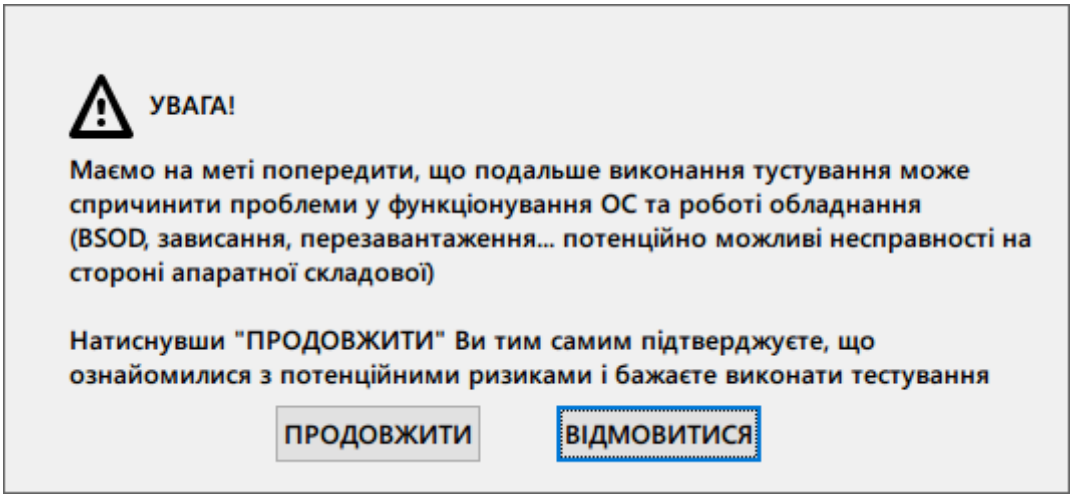


Рис. 2.41 Вікно тестування з попередженням

Тестування є комплексним і передбачає застосування різних типів атак на систему. Оскільки досліджувані технології схожі за функціональністю, то атаки

можна розділити на дві основні групи (див. рис. 2.42). Інформація стосовно цього міститься у першому вікні Процесу Тестування.

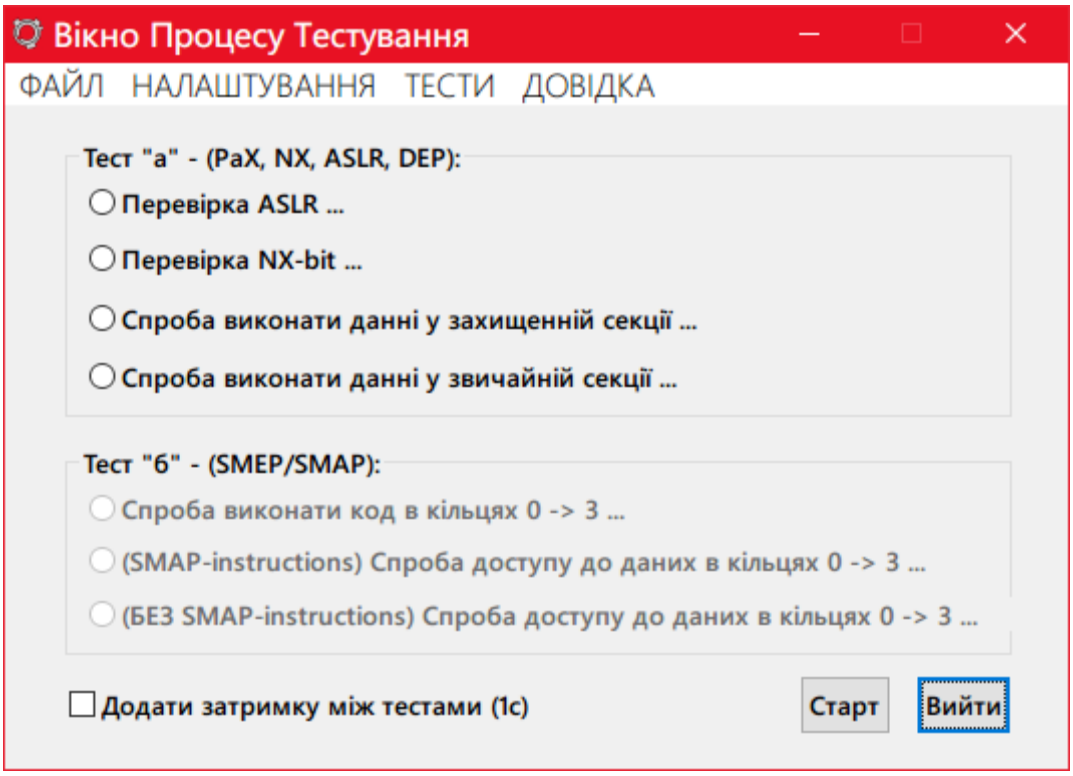


Рис. 2.42 Вікно Процесу Тестування з налаштуваннями параметрів

Наприклад, можна обрати наступні параметри тестувань:

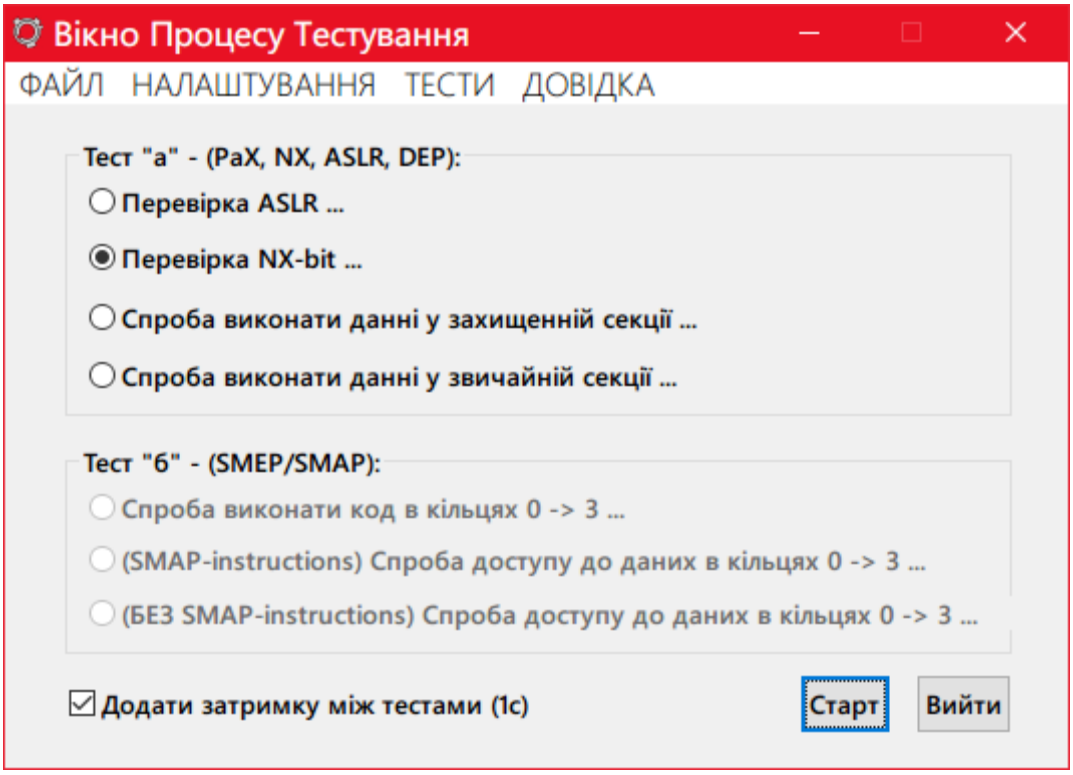


Рис. 2.43 Вікно Процесу Тестування з налаштуваннями параметрів

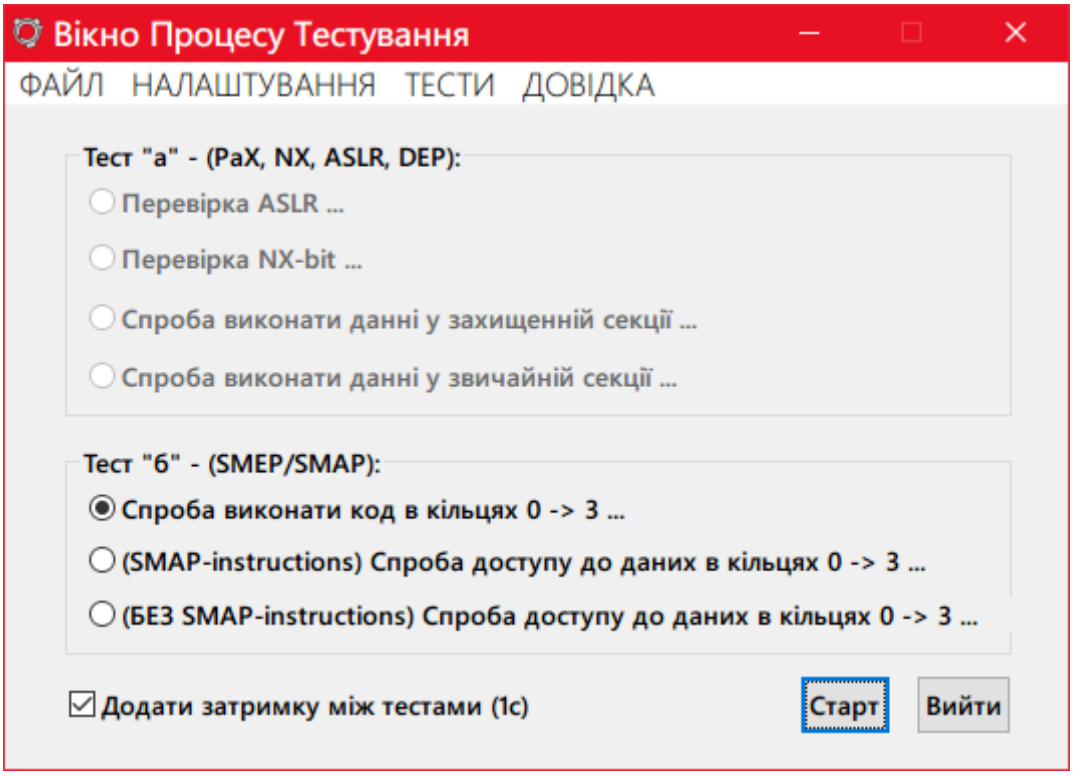


Рис. 2.44 Вікно Процесу Тестування з налаштуваннями параметрів

Наступним кроком користувача є натискання кнопки «Старт», яка направить його у діалогове вікно самого Тестування:

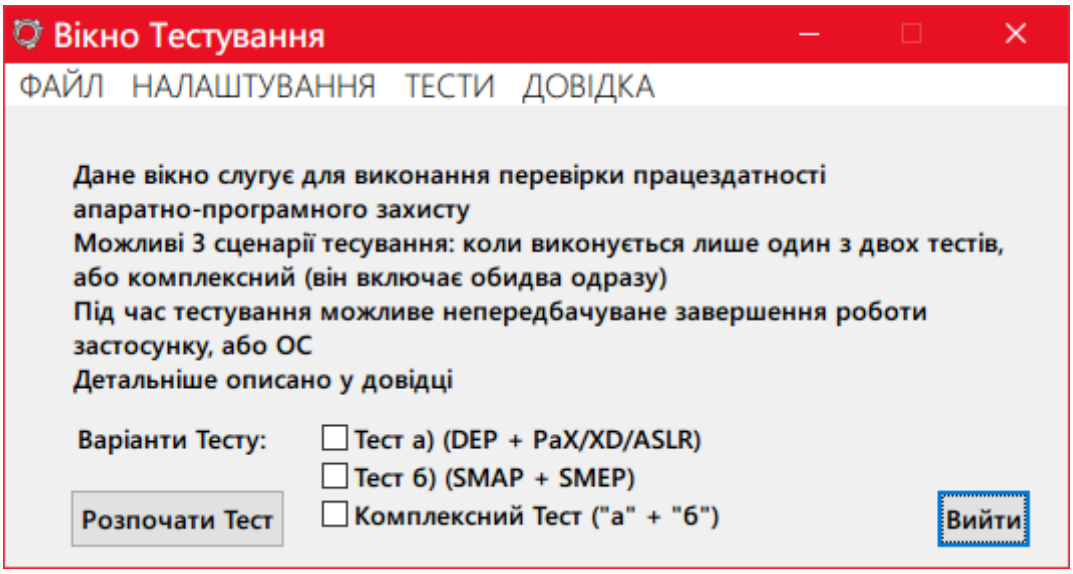


Рис. 2.45 Вікно Тестування

Так, програма передбачає можливість виконання окремих тестів (див. рис. 2.46, 2.47) або їх сукупного виконання (див. рис. 2.48).

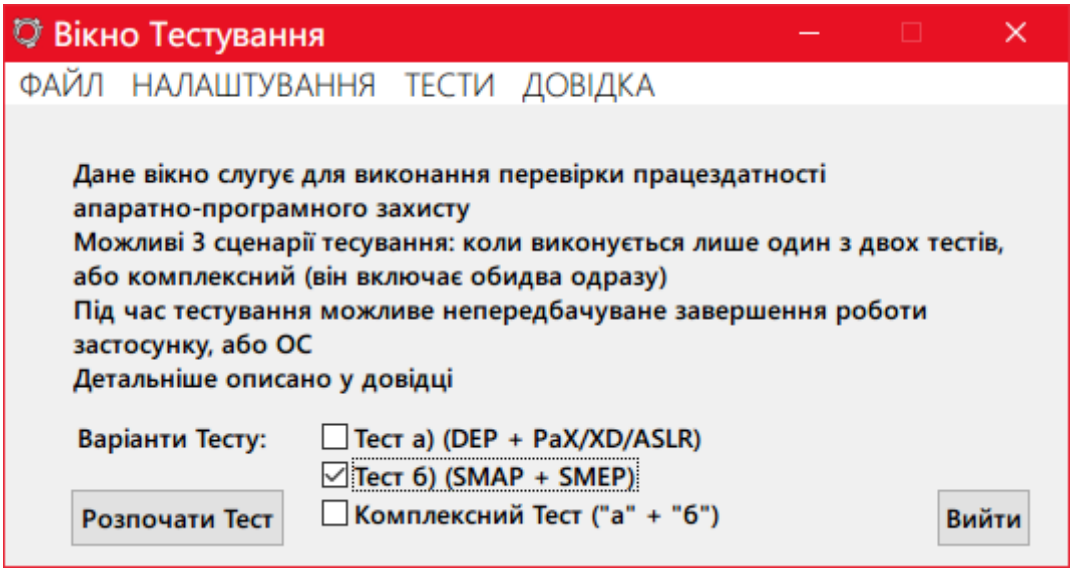


Рис. 2.46 Вікно Тестування

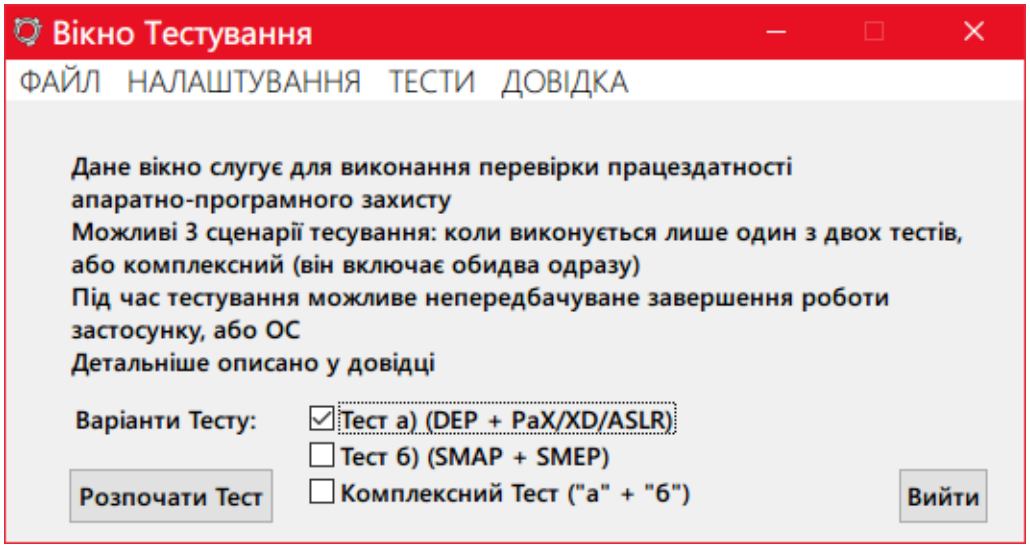


Рис. 2.47 Вікно Тестування

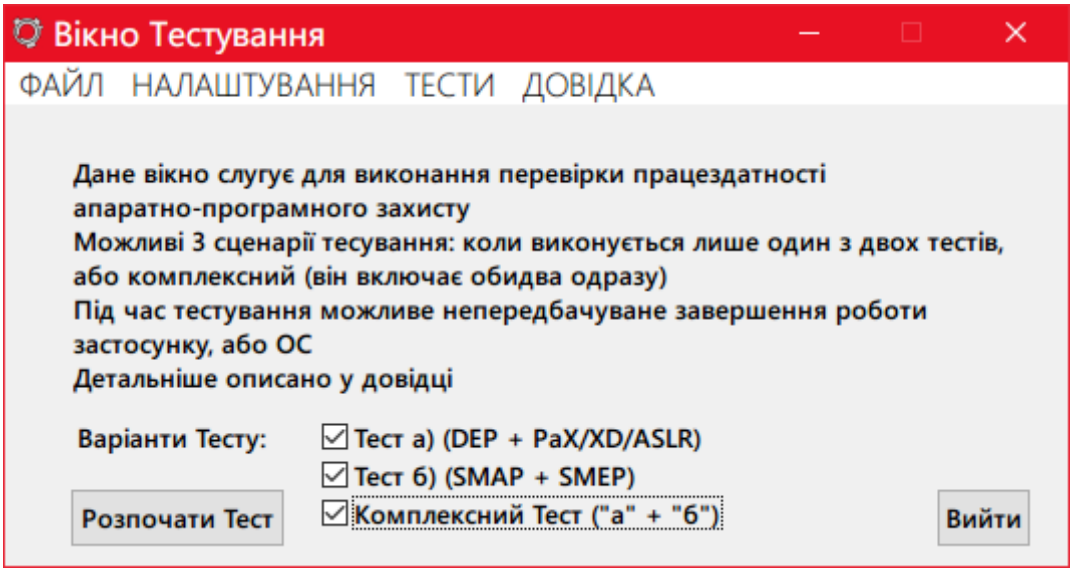


Рис. 2.48 Вікно Тестування

Після натискання кнопки «Розпочати тест», розроблений програмний додаток виконує обраний(і) тест(и). У результаті успішного виконання тестів ОС змушена буде здійснити обробку винятків, відповідно до принципу дії технологій, і завершити роботу програми або викликати BSOD. Так, наприклад, відповідно до принципу дії SMAP, у разі запиту доступу до області пам’яті користувача з-під режиму ядра (коли система перебуває у режимі супервізора), користувач бачить «синій екран смерті» про помилку. Так вийшло й у процесі практичної реалізації драйвера. Попередньо було зазначено, що один із визначних застосунків драйвера містить шкідливий код, призначений для перевірки захисного механізму технології. Якщо процесор користувача підтримує SMAP апаратно, то у момент виконання процедури, яка виконує небезпечний код, екран ПК виводить наступне повідомлення (див. рис. 2.49), змінити який задля повторної спроби отримати доступ до пам’яті ядра чи подальшої роботи у системі можливо лише через її перезавантаження.

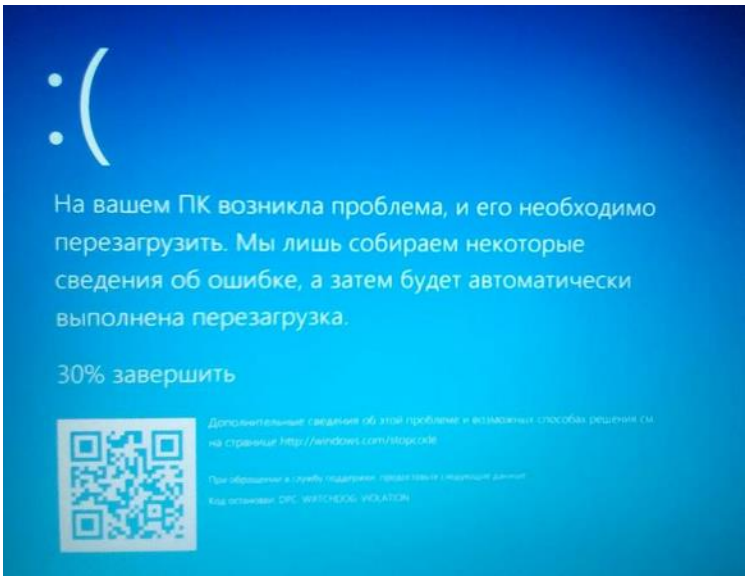


Рис. 2.49 Результат пройденого тестування системи на реакцію на атаки

Отже, програмний засіб не лише містить функції, необхідні для програмної реалізації технології SMAP у ОС Windows і забезпечення таким чином її захисту, але й успішно виконує їх на практиці при взаємодії з користувачем.

ВИСНОВКИ ДО РОЗДІЛУ 2

Другий розділ вміщує практичну частину роботи. У ній зосереджено увагу на виконанні практичних завдань і досягненні основної мети роботи – удосконаленні програмного засобу захисту ОС Microsoft Windows шляхом запровадження механізмів захисту сучасних технологій Intel. Враховуючи можливості даної ОС, засіб було розроблено у вигляді комплексу програмних застосунків-компонентів. Їх перелік, опис та основні функції викладено у другому пункті розділу.

Засіб розроблено мовою асемблера, адже вона надає можливість не тільки безпосередньо виконувати різноманітні інструкції процесора з найбільшою швидкістю та мінімальним навантаженням, а й дослідити і скеровувати процес створення програмного засобу. Це дозволило уникнути використання громіздкого програмного середовища для створення драйверів, а також надало змогу керувати змістом засобу. Такий контроль вирішив одну з головних проблем розроблення драйверів під ОС Windows 10 – зміну дизайну програмного середовища для функціонування коду на рівні драйвера.

Коли драйвер завантажується, то виконує необхідні команди для активації технологій на вимогу користувача і використовує вбудований системний гучномовець. Те ж саме відбувається і при деактивації технологій драйвером. Гучномовець низьких частот (зазначений попередньо гучномовець використовував високі частоти) же визначений сповіщати про помилку. Так, саме його чує користувач при виведенні на екран BSOD («синього екрану смерті») під час перевірки на практиці захисного механізму. Це передбачено у якості нормальної поведінки засобу з огляду на те, що помилка виникає унаслідок виконання шкідливого коду, на що реагує операційна система.

Виконання основного файлу програмного засобу запускає програму, виводячи її діалогові вікна на екран для взаємодії з користувачем. Так, програма містить привітання, меню, що складається з основних чотирьох компонентів,

перехід на конкретні процедури – Налаштування і Тести. Окрім того, міститься кнопка виходу з програми. Програмний додаток також передбачає виведення на екран попередження користувача перед здійсненням тестування захисту системи внаслідок активації технологій.

Фактично, засіб надає користувачеві привілейовані повноваження у ОС, внаслідок чого стає можливим програмна активація (як і деактивація) технологій шляхом виконання визначених інструкцій, які здійснюють перемикання відповідних бітів мікропроцесора. Далі всі звернення до адресного простору пам’яті користувача контролюються і регулюються активованими технологіями, що у результаті забезпечує подальший захист системи від несанкціонованого доступу.

Також у цьому розділі продемонстровано практичну реалізацію засобу, результат якої зображено на відповідних рисунках другого пункту розділу. Внаслідок виконаного на практиці тестування системи було отримано підтвердження ефективності забезпечуваного засобом захисту, особливо у порівнянні з результатами тестування попередньо впровадженого засобу на базі лише технології SMAP. На основі отриманих даних було розраховано ефективність захисту у відсотковому еквіваленті та побудовано порівняльні графіки. На основі отриманих результатів як доказів успішного розроблення та реалізації програмного засобу, можна дійти висновку, що мету дипломної роботи було успішно досягнуто.

РОЗДІЛ 3. ОЦІНКА ЕФЕКТИВНОСТІ ЗАХИСТУ ПРОГРАМНИМ ЗАСОБОМ

Задля досягнення кінцевої мети роботи необхідно, окрім розроблення засобу, також випробувати його на ефективність. Оскільки засіб передбачає виконання тестів захисту системи, то цей процес і було обрано за фундамент формування кінцевих статистичних результатів, на основі яких можна буде здійснити оцінку ефективності розробки.

Таким чином, після активації усіх технологій було проведено декілька тестів:

1) тест «a.1» – перевірка технології ASLR шляхом застосування атаки типу «переповнення буфера» за допомогою впровадження у захищені ділянки пам’яті двійкового шелл-коду (див. рис. 3.1);

2) тест «a.2» – перевірка технології XBD шляхом застосування шелл-коду до захищених сторінок пам’яті;

3) тест «a.3» – перевірка активації технологій PaX і DEP шляхом застосування шкідливого коду до захищених сторінок пам’яті;

4) тест «a.4» – перевірка коректної роботи всіх технологій, зазначених вище, шляхом застосування шкідливого коду до звичайних ділянок пам’яті;

5) тест «b.1» – перевірка коректної роботи технології SMEP шляхом застосування атаки типу «перевищення привілеїв» внаслідок виконання коду у ділянках пам’яті, що «належать» режиму супервізора;

6) тест «b.2» – перевірка роботи SMAP внаслідок розміщення шкідливого коду у привілейованих ділянках пам’яті;

7) тест «b.3» – перевірка роботи всіх запроваджених технологій, окрім SMEP, при розміщенні коду зловмисника у привілейованих ділянках пам’яті;

8) комплексний тест «(a+ b)», що включає усі перелічені;

9) комплексний тест «(a+b)» при деактивації кожної з технологій по черзі.

```
mov QWORD PTR 48[rsp], rax
mov ecx, 8
call    malloc
mov QWORD PTR 56[rsp], rax
mov ecx, 16
call    malloc
mov QWORD PTR 64[rsp], rax
mov ecx, 16
call    malloc
mov QWORD PTR 72[rsp], rax
mov ecx, 1024
call    malloc
mov QWORD PTR 80[rsp], rax
mov ecx, 1024
call    malloc
mov QWORD PTR 88[rsp], rax
mov ecx, 16384
call    malloc
mov QWORD PTR 96[rsp], rax
mov ecx, 16384
call    malloc
mov QWORD PTR 104[rsp], rax
lea rax, [global_initialized_var]
```

```
mov QWORD PTR 40[rsp], rax
lea edx, [rax]
lea rcx, .LC0[rip]
call    test_base
lea rax, [static_initialized_var]
mov QWORD PTR 40[rsp], rax
lea edx, [rax]
lea rcx, [rip]
call    test_base
lea rax, [global_var]
mov QWORD PTR 40[rsp], rax
lea edx, [rax]
lea rcx, [rip]
call    test_base
lea rax, [static_var]
mov QWORD PTR 40[rsp], rax
lea edx, [rax]
lea rcx, [rip]
call    test_base
mov edx, ebx
lea rcx, [rip]
call    test_base
mov ebx, 0
```

Рис. 3.1 Фрагменти коду а (зліва) та б (справа) тестування технології ASLR

Як було зазначено раніше, у результаті успішного виконання тестів ОС змушена буде завершити роботу програми або викликати BSOD. У разі відсутності такої реакції – тест не пройдено, тобто, шкідливий код був виконаний, не дивлячись на прогнозований захист технологіями. При комплексному тестуванні у разі, коли один будь-який тест зазнає невдачі, – всі інші тести з послідовності перериваються. Це нормальна робоча поведінка програми.

Тож, внаслідок виконання тестування, було отримано наступні результати:

Таблиця *2.1*

Результати тестування програмного засобу на базі сучасних технологій Intel

Тест	PaX		XD		ASLR	
	Вкл.	Викл.	Вкл.	Викл.	Вкл.	Викл.
a.1	Пройдено	Невдача	Пройдено	Невдача	Пройдено	Невдача
a.2	Пройдено	Невдача	Пройдено	Невдача	Пройдено	Невдача
a.3	Пройдено	Невдача	Пройдено	Невдача	Пройдено	Пройдено
a.4	Пройдено	Невдача	Пройдено	Невдача	Пройдено	Пройдено
b.1	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено

Продовження таблиці *2.1*

b.2	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено
b.3	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено
(a+b)	Пройдено	Невдача	Пройдено	Невдача	Пройдено	Невдача
Тест	DEP		SMEP		SMAP	
	Вкл.	Викл.	Вкл.	Викл.	Вкл.	Викл.
a.1	Пройдено	Невдача	Пройдено	Пройдено	Пройдено	Пройдено
a.2	Пройдено	Невдача	Пройдено	Пройдено	Пройдено	Пройдено
a.3	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено
a.4	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено	Пройдено
b.1	Пройдено	Пройдено	Пройдено	Невдача	Пройдено	Невдача
b.2	Пройдено	Пройдено	Невдача	Невдача	Пройдено	Невдача
b.3	Пройдено	Пройдено	Невдача	Невдача	–	Невдача
(a+b)	Пройдено	Невдача	Невдача	Невдача	Пройдено	Невдача

Отже, у більшості випадків здійснення атак, вони були успішно відбиті операційною системою після активації зазначених технологій. Негативні результати у випадку тестів для PaX, XD, ASLR і DEP ще раз наголошують на суміжності технологій, а отже і принципу дії та, відповідно, недоліків. На основі отриманих результатів було побудовано наступну діаграму:

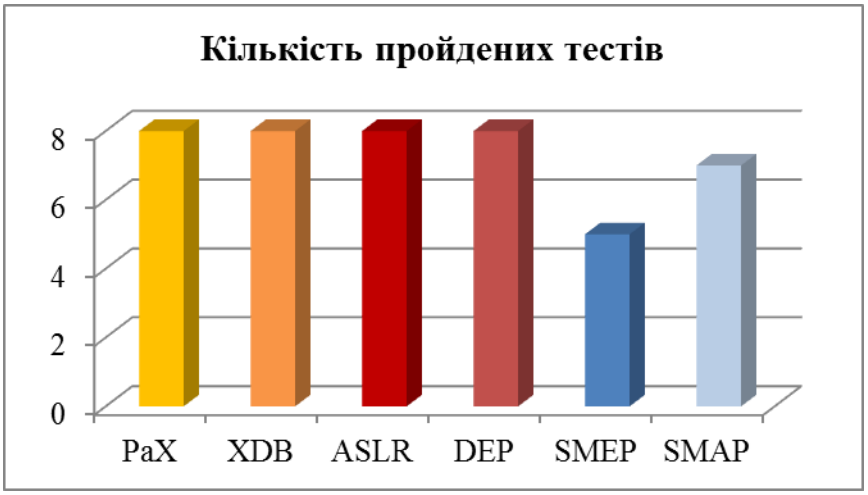


Рис. 3.2 Кількість пройдених тестів кожною з впроваджених технологій

Також можна помітити, що не всі технології, запроваджені у якості вдосконалення програмного засобу, однаково успішно впоралися з атаками. Так, атаки типу переповнення буфера були краще відбиті, ніж атаки типу перевищення привілеїв у випадку деактивації технологій SMAP або SMEP і навіть активації останньої. Відповідно, далі можна побачити діаграму успішних результатів для кожного тесту (див. рис. 3.3-3.6):

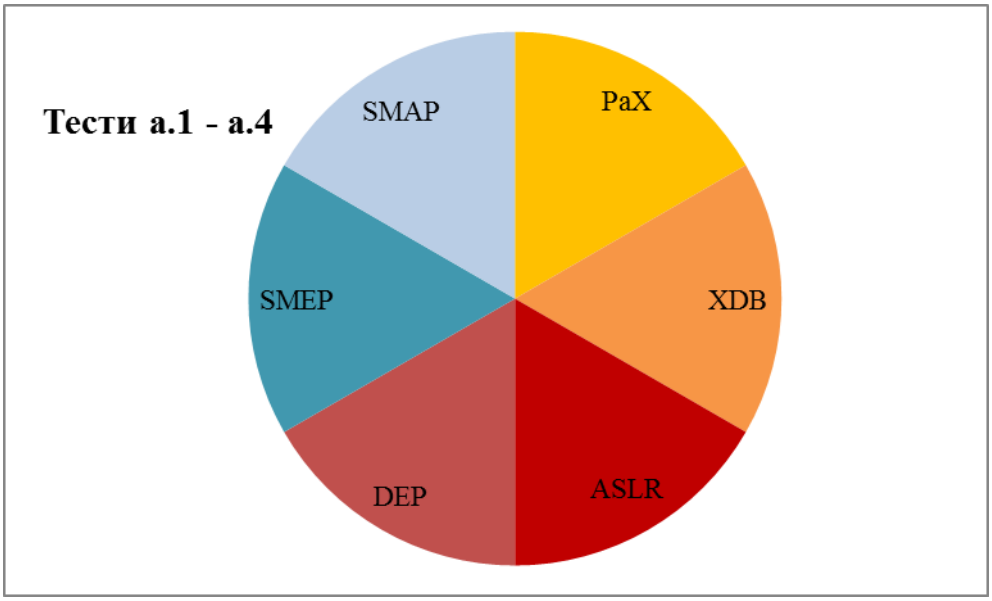


Рис. 3.3 Успішність спрацювання захисних механізмів технологій за результатами тестів a.1 - a.4

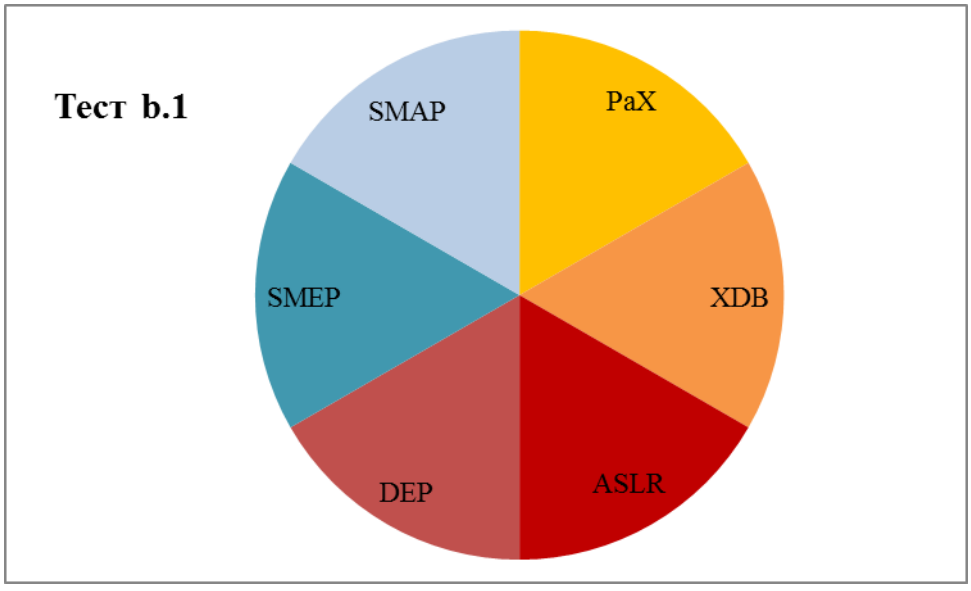


Рис. 3.4 Успішність спрацювання захисних механізмів технологій за результатами тесту b.1

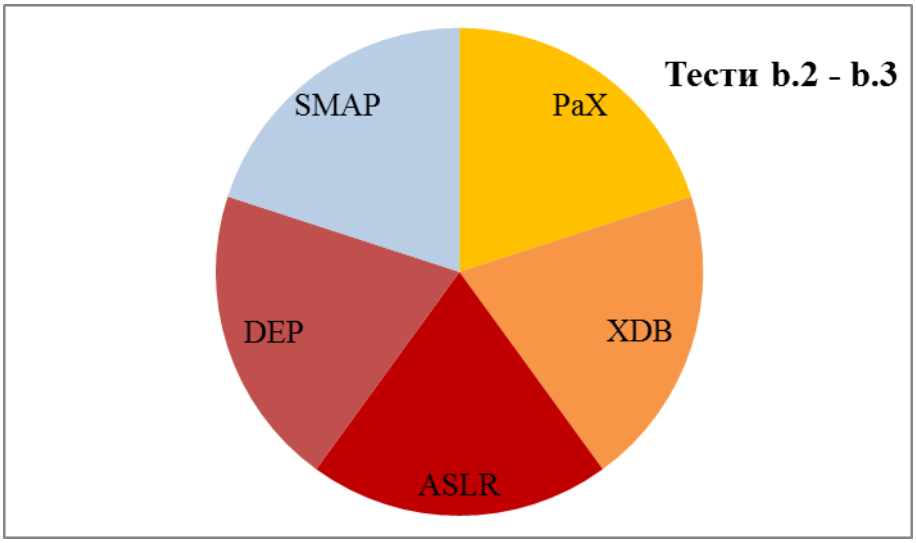


Рис. 3.5 Успішність спрацювання захисних механізмів технологій за результатами тестів b.2 - b.3

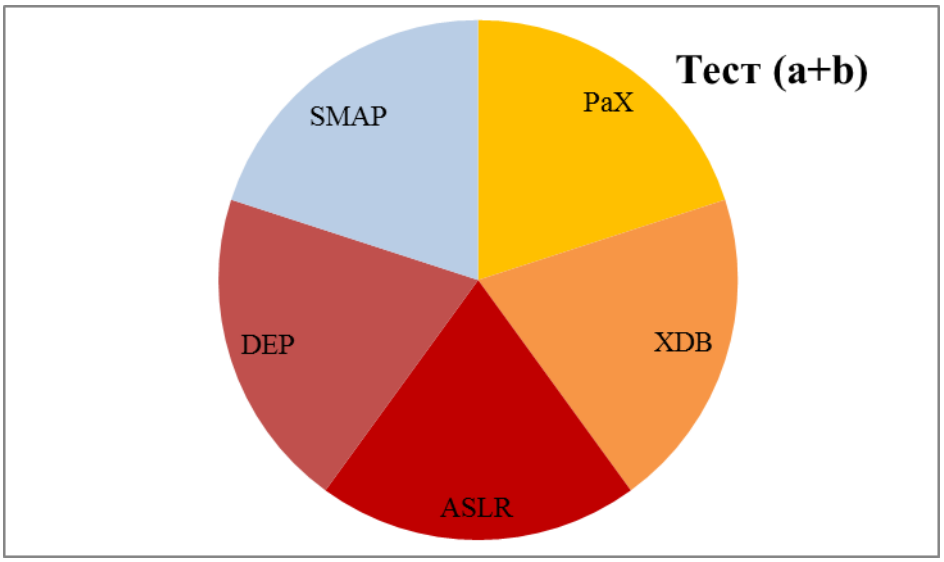


Рис. 3.6 Успішність спрацювання захисних механізмів технологій за результатами комплексного тесту (a+b)

Задля порівняння указані тести були проведені незалежно за допомогою коду також для початкового програмного драйвера на базі лише технології SMAP. Результати наведено у табл. 2.2.

Таблиця *2.2*

Результати тестування програмного засобу на базі Intel SMAP

Тест	SMAP	
	Вкл.	Викл.
a.1	Невдача	Невдача

Продовження таблиці *2.2*

a.2	Невдача	Невдача
a.3	Пройдено	Невдача
a.4	Невдача	Невдача
b.1	Пройдено	Невдача
b.2	Пройдено	Невдача
b.3	–	Невдача
(a+b)	Невдача	Невдача

Результат застосованого до драйвера тесту наведено на наступній діаграмі (де повністю успішне проходження тестів технологією рівне 8 успішним тестам, через тест b.3, який виконується без застосування інструкцій технології):

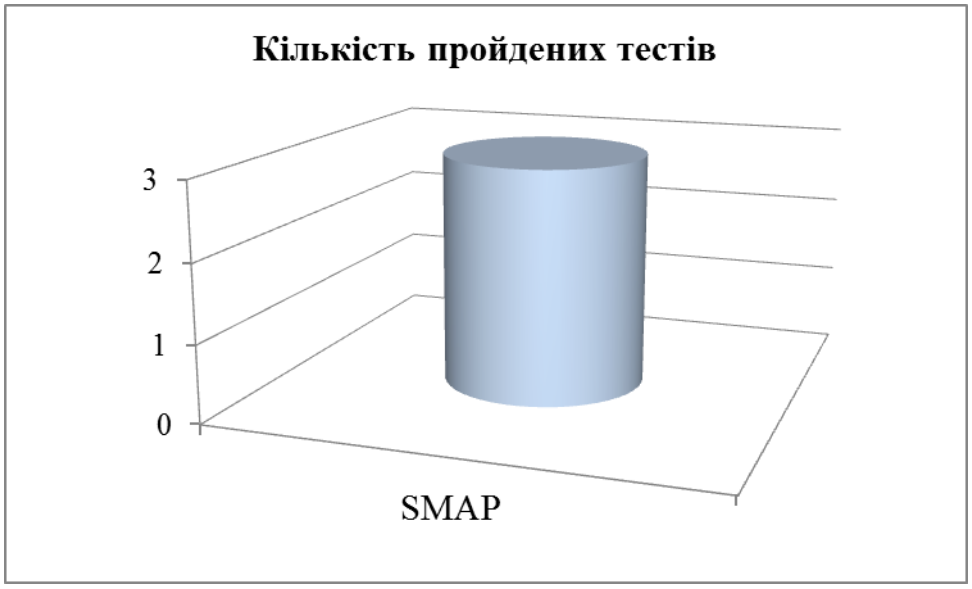


Рис. 3.7 Кількість пройдених тестів лише технологією SMAP

Тож, помітно, що надання захисту лише на основі SMAP має сенс, проте не є ефективним у порівнянні із захистом внаслідок впровадження декількох технологій захисту. Розгортання декількох видів захисних механізмів надійніше захищає від різних експлойтів: там, де одна технологія захисту зазнає невдачі, інша виявиться запобіжною.

Опрацьовані дані дозволяють оцінити ефективність впровадження удосконаленого програмного засобу шляхом обчислення зміни рівня захищеності

системи за його рахунок. Таким чином, у даному випадку, якщо взяти за забезпечення 100%-го рівня захищеності проходження всіх тестів з відбиванням атак, то можна прослідкувати наступну залежність рівня захищеності від кількості впроваджених технологій забезпечення захисту системи:

Рис. 3.8 Рівень захищеності операційної системи у залежності від впроваджених технологій

На основі графіку можна дійти висновку, що у даному випадку більша кількість впроваджених технологій захисту дійсно відповідає вищому рівню захищеності: за рахунок проведених удосконалень він підвищився у втричі. При цьому варто зазначити, що дані результати справедливі лише у даному конкретному випадку з огляду на відносну універсальність технології SMAP і взагалі впроваджені технології, адже кожна окрема технологія має свої слабкі місця, а тому змінює рівень захищеності (що було доведено на основі зведеної таблиці результатів тестування розробленого програмного засобу).

Стосовно створюваного навантаження на систему при виконанні засобу: вона є мінімальною завдяки його монолітності. Проте навіть так удосконалений засіб показує свою перевагу над попереднім у продуктивності, оскільки за спостереженнями один окремий тест ОС на захищеність шляхом взаємодії з попе-

реднім програмним засобом триває у середньому ~ 1.00 секунду, тоді як при взаємодії з актуальним – ~ 0.01 секунди.

Отже, ефективність удосконаленого програмного засобу у порівнянні з попереднім успішно доведено. За результатами оцінки забезпечуваного засобом захисту можна зробити висновок про бажаність його подальшого вдосконалення задля зведення до мінімуму ймовірності пропуску атак.

ВИСНОВКИ ДО РОЗДІЛУ 3

Здійснивши власне дослідження у вигляді тестування операційної системи після запровадження у її середовищі удосконаленого програмного засобу захисту та порівнявши його результати з аналогічним тестуванням попередньо розробленого програмного драйвера, спрямованого також на захист операційної системи, було одержано результати, що дали змогу дійти наступних висновків: надання захисту лише на основі однієї програмно-апаратної технології захисту (у даному випадку SMAP) має сенс, проте не є ефективним у порівнянні із захистом внаслідок впровадження декількох технологій. Розгортання декількох видів захисних механізмів надійніше захищає від різних експлойтів: там, де одна технологія захисту зазнає невдачі, інша виявиться запобіжною.

Опрацьовані дані дозволяють оцінити ефективність впровадження удосконаленого програмного засобу шляхом обчислення зміни рівня захищеності системи за його рахунок. Таким чином, можна прослідкувати таку залежність рівня захищеності від кількості впроваджених технологій захисту системи, при якій більша кількість впроваджених технологій дійсно відповідає вищому рівню захищеності: за рахунок проведених удосконалень він підвищився у втричі. При цьому варто зазначити, що дані результати справедливі лише у даному конкретному випадку з огляду на відносну універсальність технології SMAP і взагалі впроваджені технології, адже кожна окрема технологія має свої слабкі місця, а тому змінює рівень захищеності (що було доведено на основі зведеної таблиці результатів тестування розробленого програмного засобу).

Стосовно створюваного навантаження на систему при виконанні засобу: вона є мінімальною завдяки його монолітності.

Отже, ефективність удосконаленого програмного засобу у порівнянні з попереднім успішно доведено. За результатами оцінки забезпечуваного засобом захисту можна зробити висновок про бажаність його подальшого вдосконалення задля зведення до мінімуму ймовірності пропуску атак.

ВИСНОВКИ

У дипломній роботі висвітлено проблему захисту операційної системи Windows від несанкціонованого доступу із метою збереження інформаційних ресурсів від несанкціонованого втручання. У першому розділі доведено, що у нагоді вирішенню даної проблеми може сприяти впровадження сучасних апаратних технологій, використовуваних процесорами корпорації Intel, а саме SMEP, SMAP, XDB, а також PaX, DEP, ASLR. Загалом, вони мають схожий принцип дії – позначення необхідних ділянок пам'яті недоступними для небажаних додатків чи процесів, зводячи їх спроби на отримання доступу до обробки винятків. Це дає змогу захистити операційну систему від багатьох експлойтів, зокрема, тих, що користуються вразливостями типу «переповнення буфера пам'яті», «повернення в бібліотеку», «перевищення привілеїв» і навіть експлойтів механізмів самої обробки винятків.

Надання захисту операційній системі шляхом впровадження сукупності зазначених технологій не має жодної програмної реалізації, окрім як у вигляді модуля ядра, що малося на меті виправити у даній роботі шляхом розроблення, демонстрації роботи та аналізу ефективності впровадження засобу забезпечення захисту операційної системи Windows.

Задля досягнення кінцевої мети у процесі виконання магістерської роботи було проведено:

– дослідження сучасних технологій Intel, які задовольняють вимоги забезпечення захисту виконуваного простору, у тому числі принципу їх дії, реалізації та підтримки, а також супутніх понять;

– удосконалення програмного засіб захисту ОС MS Windows на основі Intel SMAP шляхом впровадження механізмів захисту досліджуваних технологій;

– тестування та оцінку ефективності впроваджених удосконалень.

– порівняльний аналіз захищеності ОС MS Windows, забезпечуваної засобом на базі досліджуваних технологій Intel та засобом на базі технології Intel SMAP.

Це дало змогу виконати:

- Детальне вивчення досліджуваних сучасних технологій Intel, опрацювання досліджених даних, їх структурування, компонування і систематизацію.
- Визначення вимог до програмного засобу, серед яких – перелік основних необхідних виконуваних засобом функцій на етапі його планування і визначення його структури. Обрання доцільних технологій для розроблення програмного засобу.
- Розроблення основних компонентів програмного засобу у вигляді драйвера. Розроблення сценаріїв для наглядної демонстрації роботи засобу, створення рекомендацій та інструкції до них.
- Порівняльний аналіз захищеності ОС MS Windows на базі всіх досліджуваних технологій та лише однієї з них із розрахунком ефективності захисту у відсотковому еквіваленті та побудові порівняльних графіків

Результатом роботи є програмний засіб захисту операційної системи MS Windows на базі сучасних апаратних технологій Intel, реалізований у якості програмно-апаратного драйвера, виконаного із застосуванням мов програмування C++ та Assembler для 64-розрядної версії ОС Windows 10.

Розроблений засіб вперше дозволяє програмно керувати сукупністю зазначених технологій шляхом взаємодії з «закритим» ядром ОС Windows та мікропроцесором, надаючи користувачеві привілейовані повноваження у системі. Засіб також контролює і регулює звернення до адресного простору пам’яті користувача і пам’яті ядра, що і забезпечує подальший захист ресурсів системи від несанкціонованого доступу. До того ж він надає можливість здійснити тестування захищеності системи із метою подальшого її вдосконалення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Загальні положення із захисту інформації в комп'ютерних системах від несанкціонованого доступу [Текст] : НД ТЗІ 1.1-002-99 – 1999. – Чин. 1999-07-01. – К. : ДСТСЗІ СБ України, 1999. – 21 с.

2. Інформаційні технології. Класифікація програмних засобів: ДСТУ ISO/IEC TR 12182: 2004, IDT. – 2004. Введ.: 01.01.2006. – К.: Держстандарт України, 2006. – 14 с.

3. Інформаційні технології. Процеси життєвого циклу системи: ДСТУ ISO/IEC 15288:2005. – 2005. Введ.: 01.01.2007. – К.: Держстандарт України, 2005. – 20 с.

4. Концепція технічного захисту інформації в Україні [Текст]: Постанова Кабінету Міністрів України про затвердження Концепції технічного захисту інформації в Україні № 1126 від 08.10.97 р. / Кабінет Міністрів України, 2008.

5. Положення про технічний захист інформації в Україні [Текст]: Указ Президента України про Положення про технічний захист інформації в Україні № 1229/99 від 27.09.99 р. / Президент України, 2008.

6. Порівняння випусків і версій Windows 10 // Home і Pro. – Microsoft Corp., 2018.

7. Про захист інформації в інформаційно-телекомунікаційних системах [Текст] : Закон України №80/94-ВР від 05.07.94 р. / Верховна Рада України // Відомості Верховної Ради України. – 2014. – №31. – Ст. 287.

8. Про захист персональних даних [Текст] : Закон України № 2297-VI від 01.06.2010 р. / Верховна Рада України // Відомості Верховної Ради України. – 2010. – №34. – Ст. 481.

9. Про інформацію [Текст]: Закон України №2657-XII від 02.10.92 р. / Верховна Рада України // Відомості Верховної Ради України. – 1992. – №48. – Ст. 650.

10. Про основні засади забезпечення кібербезпеки України [Текст] : Закон України № 2469-VIII від 21.06.2018 р. / Верховна Рада України // Відомості Верховної Ради України. – 2017. – №45. – Ст. 403.

11. Системи оброблення інформації. Програмування. Терміни та визначення: ДСТУ 2873-94 – 1996. Введ.: 01.01.1996. – К.: Держстандарт України, 1994. – 39 с.

12. Системи оброблення інформації. Розроблення систем. Терміни та визначення: ДСТУ 2941-94 – 1996. Введ.: 01.01.1996. – К.: Держстандарт України, 1994. – 19 с.

13. Термінологія в галузі захисту інформації в комп'ютерних системах від несанкціонованого доступу [Текст] : НД ТЗІ 1.1-003-99 – 1999. – Чин. 1999-07-01. – К. : ДСТСЗІ СБ України, 1999. – 30 с.

14. Харт Джонсон М. Системное программирование в среде Windows / Джонсон М. Харт ; пер. с англ. — М. : Издательский дом «Вильямс», 2005.

15. Єдина система програмної документації. Настанова системного програміста. Вимоги до змісту та оформлення: ГОСТ 19.503-79 (СТ СЗВ 2094-80). – 1980.

16. A DEP evasion technique. [Електронний ресурс] // blog on cyberterror – Електр. дані – 2005. – Режим доступу до ресурсу: World Wide Web. – URL: http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html

17. Belousov K. Revision 242433 / K. Belousov // Information Security Handbook. – New York: Stockton Press, 2018, №13, P. 25-29.

18. Belousov K. Revision 242476 / K. Belousov // Information Security Handbook. – New York: Stockton Press, 2018, №16, P. 17-20.

19. Burke T. Writing device drivers: tutorial and reference / Timothy Burke // Digital Press, 1995.

20. Corbet J. Supervisor mode access prevention [Електронний ресурс] / Jonathan Corbet // LWN.net. – Електр. Дані – 2017. – Режим доступу до ресурсу: World Wide Web. – URL: https://lwn.net/Articles/342330/.

21. Gruss D. Prefetch Side-Channel Attacks: Bypassing SMAP and kernel ASLR / Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, Stefan Mangard // G DATA Advanced Analytics, CCS'16 — Graz University of Technology, 2016.

22. Dijkstra E.W. Structured Programming / E. Dijkstra // Software Engineering Techniques, Buxton, J.N., and Randell, B. NATO Science Committee – Brussels, Belgium, 1969.

23. Hewlett Packard Data Execution Prevention v1.2 // ® HP Inc., 2005.

24. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4 // ® Intel Corporation, 2019.

25. Johnson P. Yasm User Manual, win32: Safe Structured Exception Handling / Peter Johnson //Tortall Networks: Open Source and Free Software, 2015.

26. Larabel M. Intel SMAP Comes To Try To Better Secure Linux / Michael Larabel // Phoronix, 2017.

27. Mateusz Jurczyk SMEP: What is it, and how to beat it on Windows / Mateusz "j00ru" Jurczyk, Gynvael Coldwind, 2011.

28. Mateusz Jurczyk Windows Security Hardening Through Kernel Address Protection / Mateusz "j00ru" Jurczyk, 2011.

29. Microsoft Win32 Memory Management // © Microsoft Corporation, 2018.

30. Microsoft Security Intelligence Report (SIR) Volume 21 // © Microsoft Corporation, 2016.

31. Sandeep Bhatkar, R. Sekar. Efficient techniques for comprehensive protection from memory error exploits / Sandeep Bhatkar, R. Sekar, Daniel C. DuVarney // USENIX Security Symposium, 2005.

32. Villard M. AMD64: SMAP support [Електронний ресурс] / Maxime Villard // tech-kern archive – Електр. дані – 2018. – Режим доступу до ресурсу: World Wide Web. – URL: https://mail-index.netbsd.org/tech-kern/2017/08/23/msg022249.html.

33. OpenBSD 5.3. [Електронний ресурс] // OpenBSD – Електр. дані – 2017. – Режим доступу до ресурсу: World Wide Web. – URL: https://www.openbsd.org/53.html.

34. Fischer S. Supervisor Mode Execution Protection / Stephen Fischer // NSA Trusted Computing Conference, 2011.

35. Windows Hypervisor Platform API [Електронний ресурс] // Micorosft Docs, Virtualization – Microsoft Corp. – Електр. дані – Режим доступу до ресурсу: World Wide Web. – URL: https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/hypervisor-platform.

36. x64 (amd64) intrinsics list [Електронний ресурс] // Micorosft Docs, Compiler intrinsics – Microsoft Corp. – Електр. дані – Режим доступу до ресурсу: World Wide Web. – URL: https://docs.microsoft.com/en-us/cpp/intrinsics/x64-amd64-intrinsics-list?view=vs-2019.