КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№2 (6), 2019
ISSN 2663 - 4023
DOI: 10.28925/2663-4023.2019.6.8293
УДК 004.49

Гречко Вікторія Володимирівна
бакалавр, студентка кафедри кібербезпеки та захисту інформації
Київський національний університет імені Тараса Шевченка, факультет інформаційних технологій, Київ, Україна
ORCID ID: 0000-0002-5190-4742
grechko.viktoria@gmail.com

Бабенко Тетяна Василівна
доктор технічних наук, професор, професор кафедри кібербезпеки та захисту інформації факультету інформаційних технологій
Київський національний університет імені Тараса Шевченка, факультет інформаційних технологій, Київ, Україна
ORCID ID: 0000-0003-1184-9483
babenkot@ua.fm

Мирутенко Лариса Вікторівна
кандидат технічних наук, доцент кафедри кібербезпеки та захисту інформації
Київський національний університет імені Тараса Шевченка, факультет інформаційних технологій, Київ, Україна
ORCID ID: 0000-0003-1686-261X
myrutenko.lara@gmail.com

БЕЗПЕЧНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ЩО РОЗРОБЛЯЄ РЕКОМЕНДАЦІЇ

Анотація. Шкідливий вплив на інформацію в процесі функціонування комп'ютерних систем різного призначення здійснюється з метою порушення конфіденційності, цілісності і доступності і є можливим внаслідок експлуатації наявних вразливостей. Результатом такого впливу може бути несанкціонований доступ до даних або витік конфіденційної інформації. Актуальність розробки рекомендацій по створенню безпечного програмного забезпечення (ПЗ) полягає у вдосконаленні підходів до розробки ПЗ з метою ліквідації вразливостей для нового ПЗ та досліджень вже створеного ПЗ на предмет відсутності в ньому вразливостей. Для вирішення цієї проблеми, по-перше, було проведено аналіз життєвих циклів програмного забезпечення з метою визначення основних етапів його розробки. Наступним кроком були визначені можливі загрози для інформації на кожному з етапів розробки. Розглянуто уразливість переповнення буферу як приклад. Наведено можливі способи експлуатації цієї вразливості, проаналізовано переваги і недоліки засобів виявлення та протидії. Як результат, запропоновано рекомендації щодо розробки безпечного програмного забезпечення як на загальному рівні, так і більш конкретні стосовно переповнення буфера. Практичною цінністю рекомендацій є зменшення ризиків порушення властивостей інформації, що підлягає захисту, і мінімізація витрат організації. Отримані в роботі результати також можуть бути використані для прийняття рішень про можливість експлуатації відповідного програмного забезпечення.

Ключові слова: розробка безпечного програмного забезпечення; життєвий цикл розробки програмного забезпечення; вразливості; переповнення буфера; статичний та динамічний аналіз; механізми запобігання переповнення буферу.

ISSN 2663 - 4023

1. ВСТУП

Шкідливий вплив на інформацію в процесі функціонування комп'ютерних систем різного призначення здійснюється з порушенням її конфіденційності, цілісності та доступності. У той же час, розглядаючи інформацію як активний експлуатований ресурс, можна сказати, що процес забезпечення безпеки інформації включає і безпеку програмного забезпечення комп'ютерних систем. Вирішення задач, пов'язаних з запобіганням впливу безпосередньо на інформацію здійснюється в рамках комплексної задачі забезпечення інформаційної безпеки.

Оскільки роль техніки продовжує посилюватись у кожному аспекті суспільства, пошук методів зменшення кількості вразливостей у кінцевому продукті є більш критичним, ніж будь-коли.

Постановка проблеми. Для збереження рівноваги між прагненням до високого рівня безпеки та функціональністю, сервіси безпеки повинні проектуватися і розроблятися одночасно з функціональністю та забезпечуватися на всіх рівнях. В іншому випадку безпека забезпечується не в повному обсязі, залишаючи істотні уразливості. Побудова надійного захисту комп'ютерної системи неможлива без попереднього аналізу можливих загроз безпеці інформації, що циркулює в системі, порівняння існуючих моделей життєвого циклу та кращих практик з розробки безпечного програмного забезпечення. Для покращення існуючих підходів до розробки програмного забезпечення, тестування нового програмного забезпечення з метою усунення вразливостей, а також виявлення вразливостей для вже створеного необхідно розробити рекомендації по створенню безпечного програмного забезпечення.

Аналіз останніх досліджень і публікацій. В загальному випадку під загрозою інформаційній безпеці розуміється потенційно можлива подія, дія або вплив, процес або явище, які можуть призвести до нанесення шкоди ресурсам інформаційної системи. Як правило, загрози для інформаційної безпеки розрізняють за способом реалізації. Актуальна загальноприйнята класифікація дефектів за типами представлена в базі Common Weakness Enumeration [6], список зареєстрованих вразливостей - в базі Common Vulnerabilities and Exposures організації MITRE. Найбільш поширеними типами дефектів є:

- переповнення буфера;
- помилки при роботі з динамічною пам'яттю (витік пам'яті, розіменування нульових покажчиків і ін.);
- помилки обробки даних користувача;
- помилки форматуючих рядків;
- помилки синхронізації (взаємні блокування, відсутні блокування і т.п.);
- витік пам'яті;
- некоректна робота з тимчасовими файлами і іншими інтерфейсами ОС;
- вразливості безпеки (слабке шифрування, зберігання пароля в явному вигляді і т.п.), що не випливають безпосередньо з дефектів.

Модель загроз і модель порушника є вихідною інформацією для розроблення політики безпеки і проектування будь-яких систем захисту. Модель загроз безпеки ПЗ повинна являти собою офіційно прийнятий нормативний документ, яким повинні керуватися замовники і розробники програмних комплексів.

В основі розробки програмного забезпечення лежить концепція життєвого циклу. Життєвий цикл розробки програмного забезпечення – методологія проектування, створення та підтримки інформаційних та промислових систем; період часу, який

ISSN 2663 - 4023

починається в момент прийняття рішення про необхідність створення програмного продукту і закінчується в момент виведення з експлуатації[2]. Найбільш загальним представленням життєвого циклу програми є модель у вигляді базових етапів-процесів, що включає в себе:

- системний аналіз і обґрунтування вимог до ПЗ;
- попереднє (ескізне) і детальне (технічне) проектування ПЗ;
- розробка програмних компонент, їх комплексування і налагодження ПЗ в цілому;
- випробування, дослідна експлуатація та тиражування ПЗ;
- регулярна експлуатація ПЗ, підтримка експлуатації та аналіз результатів;
- супровід ПЗ, його модифікація і вдосконалення, створення нових версій.

Дана модель є загальноприйнятою і відповідає як вітчизняним нормативним документам в області розробки програмного забезпечення, так і закордонним [1-6].

Мета статті. Метою цієї статті є розробка рекомендацій щодо вдосконалення підходів до розробки програмного забезпечення з метою усунення вразливостей в ньому. У цій статті розглядаються загрози на загальному та конкретному рівні, методи їх експлуатації, механізми виявлення та протидії.

2. ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖЕННЯ

Аналіз переповнення буфера. Переповнення буфера (Buffer Overflow) - явище, яке виникає в разі, коли комп'ютерна програма записує дані за межами виділеного в пам'яті буфера. У більшості випадків переповнення буфера відбувається через некоректну роботу з даними, отриманими ззовні, і з пам'яттю, якщо відсутній надійний захист з боку підсистеми програмування та операційної системи. Навіть перший самостійно розповсюджуваний інтернет-хробак, Хробак Морріса (1988), використовував переповнення буфера в Unix-демона finger для поширення між машинами [7]. Однак, навіть тридцять один рік потому, переповнення буфера залишається джерелом проблем [8-10].

Популярність переповнення буфера пояснюється тим, що більшість мов програмування високого рівня використовує технологію стекового кадру.

Залежно від різних факторів - обраного програмістом методу резервування пам'яті, а також типу внутрішньої організації пам'яті комп'ютера, - переповнення буфера може призвести до таких наслідків:

1. Спотворення даних програми, яке може призвести до неправильного порядку її роботи. Наприклад, при роботі веб-сервера, в разі вдалої атаки, користувач отримає не необхідні дані, а помилку 404.

2. Пошкодження сегментів, що контролюють виконання програми, може привести до її некоректного завершення з помилкою «access segment violation» в Linux або «general protection fault» в Windows. Поява таких помилок вказує на те, що програма намагається отримати доступ до даних, що знаходяться за межами виділеної для неї області пам'яті.

3. Некоректне завершення роботи ОС в разі, якщо були залучені структури, що контролюють її роботу.

4. Переповнення буфера дає можливість атакуючому виконати свій код на машині жертви.

5. В результаті переповнення можуть бути зіпсовані дані, розташовані слідом за буфером (або перед ним).

ISSN 2663 - 4023

Для для Intel x386 адресний простір пам’яті поділяється на наступні сегменти [11]:

1. Купа (heap) - це область динамічно розподіленої пам'яті Область купи управляется такими операторами, як malloc() (memory allocation), realloc() і free(). Malloc() дозволяє програмісту запитувати у операційної системи необхідну кількість пам'яті для зберігання даних.

2. Стек викликів (Call Stack), зазвичай називають просто стеком. В архітектурі x86 адресація починається з нуля, в інших реалізаціях адресація може починатися в протилежному напрямку. Ця область пам'яті використовується для одночасного відстеження як поточної виконуваної функції, так і всіх попередніх функцій - тих, що були викликані, щоб потрапити в поточну функцію

3. Код — область пам'яті, де будуть зберігатися інструкції центральному процесору від скомпільованої програми. Ці інструкції генеруються компілятором, але можуть бути написані і вручну.

Прикладами атак на переповнення буферу є: руйнування стеку, перезапис вказівника фрейму, атака повернення у бібліотеку, переповнення динамічної пам’яті та перезапис вказівника функції [12].

Поширеною помилкою при програмуванні на мові С є так звана «off-by-one error», або помилка на одиницю. Найчастіше це виконується в циклі, під часу перебору елементів. Наприклад, перебір елементів масиву можна розпочати з 1, а не з 0.

Уявімо, перша локальна змінна у фреймі стека є буфером, вразливим до атак типу «off-by-one error» під час обробки призначених для користувача даних. У разі, коли між даною змінною і покажчиком фрейма немає інших даних, введений додатковий байт може переписати один байт збереженого покажчиком фрейма. У зловмисника не буде можливості виконати шелл-код, який він міг заздалегідь завантажити в буфер, використовуючи збережену адресу повернення. Однак для цього необхідно зробити лише пару додаткових кроків.

Припустимо, що переповнення відбувається в функції bad_func(), яка викликається функцією good_func(). В ході атаки зловмисник зможе змінити нижчий біт - в нашому прикладі цифру 4 - збереженого покажчика фрейма функції good_func(). Нагадаємо, що в архітектурі x86 пам'ять організована за принципом, що найбільш важливим є байт з найменшою адресою, «little endian». Уявіть, що порядок бітів в пам'яті буде іншим, «big endian». У такому випадку будь-яка зміна значення покажчика фрейма призвела б до того, що він вказав би на адресу, яка перебувала б поза межами поточного контексту виконання програми. Таким чином можна змінити адресу покажчика фрейму на такий, який привів би до виконання завантаженого шелл-коду.

Основна відмінність виконання атаки, спрямованої на переповнення буфера, що знаходиться в динамічній області, полягає в тому, що ви не знайдете покажчика фрейму або адреси повернення, який можна було б перезаписати. Тим не менше, це не захистить вас від атаки. Для реалізації цього методу необхідно буде працювати з сегментом даних і коду. Як приклад, таку атаку можна провести для підміни імені тимчасового файлу, з яким працює програма. Сховищем для тимчасових даних може бути конфігураційний файл брандмауера або системи виявлення атак. У разі такої підміни цільовий файл буде перезаписаний тимчасовими даними і система захисту може перестати функціонувати взагалі. Залежно від ступеня захисту цільової системи це може привести до серйозних наслідків.

Системи аналізу та захисту програмного забезпечення. Можливість зміни основних властивостей захищеності (доступність, цілісність, конфіденційність) інформаційних ресурсів і дестабілізації процесів функціонування інформаційно-

обчислювальних систем різного призначення за допомогою застосування зловмисниками несанкціонованих впливів деструктивного характеру (атак) на уразливості ПЗ зумовлюють гостру потребу в своєчасному виявленні дефектів (вразливостей і помилок) на етапах розробки і проектування ПЗ, перевірки відповідності їх заявленої політики безпеки і реалізації механізмів захисту [13].

Існують два основних види технологій аналізу програмного забезпечення. По-перше, використовуються системи автоматичного пошуку дефектів за допомогою статичного аналізу вихідного коду програм, які можна застосовувати на ранніх етапах розробки, що робить виправлення дефектів максимально дешевим. Такі системи мають прийнятний рівнень справжніх спрацьовувань (30-70% знайдених помилок виявляються істинними) і аналізують всі можливі шляхи виконання програми без її фактичного виконання [14-16]. Популярність даних засобів пояснюється тим, що такі методи зазвичай повністю інтегровані в цикл розробки ПЗ та можуть бути застосовані як на етапі тестування, так і на більш ранніх етапах розробки.

Методи статичного аналізу мають ряд обмежень, що не дозволяють в ряді випадків досягти високої точності аналізу. По-перше, за відсутності повного вихідного коду програми виникає невизначеність, не пов'язана безпосередньо з якістю аналізу: в залежності від властивостей недоступного при аналізі коду, деяка операція може як призводити, так і не приводити до помилки. Наприклад, для бібліотечного коду часто можлива побудова некоректного виклику з призначеного для користувача коду, який приводить до виконання некоректної операції в коді бібліотеки, але при відсутності коду цього виклику помилка в коді бібліотеки, як правило, діагностуватися не повинна.

По-друге, незалежно точності статичного аналізу при виявленні конструкцій, які можуть потенційно вказувати на уразливість, у багатьох випадках не вдається встановити, чи можливий в дійсності шлях виконання програми і вхідні дані, що призводять до помилки. Видача всіх попереджень в таких ситуаціях призводить до того, що велика їх частка виявляється помилковою, роблячи систему статичного аналізу малокорисною для багатьох додатків.

Серед статичних комерційних систем можна виділити системи Coverity Insight [14], Klocwork K9 [15], GrammaTech CodeSonar [16]. Точне судження про архітектуру і алгоритми аналізу, закладених в основу цих систем, ускладнене через їх закритість, так само як і порівняння результатів їх роботи.

По-друге, застосовуються системи динамічного аналізу бінарного коду програм, що дозволяють багаторазово запускати задану програму на автоматично генерованому наборі вхідних даних і відстежувати виникнення дефектів. Системи динамічного аналізу переглядають лише частину можливих наборів вхідних даних, але при знаходженні помилки відразу дозволяють отримати дані, на яких ця помилка проявляється (тобто не мають помилкових спрацьовувань). Застосування цих систем обмежене істотними вимогами до ресурсів і обмеженням на максимальний розмір аналізованої програми (зазвичай десятки тисяч рядків коду).

Прикладом динамічних систем можна виділити QEMU [17], Valgrind [18], KLEE [19], S2E [20], Mayhem. QEMU - емулятор процесорів і обчислювальних систем, здатний емулювати всю обчислювальну систему, в цьому випадку динамічної трансляції піддається код програми, всі бібліотеки і операційна система. Valgrind є інфраструктурою для налагодження і профілювання програм, в якій транслюється лише для користувача програма в тому ж оточенні, що і при звичайному виконанні. KLEE – інструмент для проведення символьного виконання, аналіз проводиться над внутрішнім поданням компіляторної інфраструктури LLVM [40]. S2E - система вибіркового

ISSN 2663 - 4023

символьного виконання, побудована на базі QEMU і KLEE. Mayhem - система автоматичного пошуку експлуатованих вразливостей в бінарному коді.

Існують і інші класи систем виявлення дефектів у вихідному коді програм, однак точність і необхідні для використання ресурси обмежують їх область застосування. Дані системи не отримали такого поширення, як згадані системи автоматичного пошуку дефектів на основі статичного аналізу. З цих класів систем можна згадати такі системи:

- Автоматизація експертного аудиту.
- Використання обмеженого вихідного коду.
- Перевірка коректності призначених для користувача анотацій.

Проблеми захисту програмного забезпечення в галузі контролю над його використанням і подальшим поширенням в даний час прийнято вирішувати за допомогою програмно-технічних засобів - систем захисту ПЗ. У той же час для обходу і відключення подібних систем захисту існує безліч інструментальних засобів. Виникає запитання зіставити можливості засобів захисту ПЗ з можливостями засобів їх подолання. Результати такого аналізу будуть корисні для оцінки ризиків при виробництві програмних продуктів, а так само плануванні та оцінці рівня стійкості систем захисту ПЗ.

Говорячи про механізми запобігання переповнення буфера на рівні компіляції або операційної системи для операційних систем Windows NT можна виділити дві основні категорії методів [20]. До першої категорії відносяться рандомізація адресного простору (Address space layout randomization, ASLR) і запобігання виконання даних (Data Execution Prevention, DEP). До другої категорії входять механізм розміщення «канарейок» і технологія Fortify source, котра виконує перевірки функцій на більш захищені аналоги.

Рандомізація адресного простору передбачає, що програма буде завантажуватися на різні адреси. У Windows рандомізація підтримується, починаючи з Windows Vista.

Запобігання виконання даних - механізм захисту, вбудований в різні операційні системи Windows, Linux і т.д., який забороняє програмі виконувати код з області пам'яті, поміченої як «дані». Таким чином, стек і купа стають недоступними для виконання.

«Канарейка» - це спеціальне значення, розміщене на стеці, що розділяє собою простір автоматичних локальних змінних і службові дані - адресу повернення і збережені регістри. Розміщення «канарейки» виконується в пролозі функції, в епілозі її значення порівнюється з еталоном. Їх відмінність розуміється як спрацьовування помилки переповнення буфера з загрозою псування адреси повернення та інших службових даних. Програма в такому випадку аварійно завершується, не доходячи до більш серйозних порушень безпеки.

За допомогою даних методів експлуатації вразливостей переповнення буфера стає значно складнішим, однак повністю не виключає експлуатацію. Практично для кожного методу захисту є методи, що дозволяють проексплуатувати уразливість. Наприклад, ASLR часто можна обійти, переписавши тільки частину адреси, яка не змінюється, а у випадку з DEP необхідно використовувати підхід ROP (Return Oriented Programming), використовуючи виконуваний код з уже завантажених модулів для виконання необхідних дій. Однак, загроза експлуатації залишається актуальною при виконанні ряду умов:

- Виконавчий модуль програми не може бути рандомізований.
- Наявний гаджет-трамплін, який зрушує покажчик стека на певне значення.
- ELF-файл використовує «ліниве» зв'язування.
- Наявна вразливості CWE-123.

– Наявні дані, що залежать від вхідних даних на стеку в момент передачі управління на трамплін.

Нормативно-правова база. Наразі в сфері кібербезпеки, захисту інформації і зокрема розробки безпечного ПЗ існує певний перелік документів, оформлених у вигляді корпоративних, галузевих і міжнародних стандартів, що містять вказівки для розробників ПЗ і «кращі практики», які рекомендується впроваджувати в життєвому циклі ПЗ з метою створення ПЗ з мінімально можливою кількістю вразливостей і формування середовища забезпечення оперативного усунення виявлених користувачем ПЗ проблем (вразливостей ПЗ). Розробка стандартів для відкритих систем, в тому числі і стандартів в області безпеки IT, здійснюється низкою спеціалізованих міжнародних організацій і консорціумів таких, як, наприклад, ISO, IEC, ITU-T, IEEE, IAB, WOS, ECMA, X/Open, OSF , OMG і ін. [1-5].

3. РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Таким чином для зменшення ризиків порушення властивостей інформації, що підлягає захисту, і мінімізації витрат організації необхідно розробити рекомендації щодо вдосконалення підходів до розробки програмного забезпечення з метою усунення вразливостей в ньому. Пропонується використання моделі розробки захищеного програмного забезпечення, яка впроваджує питання безпеки та конфіденційності на всіх етапах процесу розробки, допомагаючи розробникам створювати безпечне програмне забезпечення, дотримуватися вимог безпеки та скорочувати витрати на розробку. Модель використовує настанови, кращі практики, інструменти та процеси для створення більш безпечних продуктів і послуг і складається з наступних етапів та заходів:

• Ініціювання проекту:
  – Визначити концепцію проекту, потреби користувача та основні цілі безпеки продукту.
  – Початковий аналіз ризиків.
• Функціональне проектування і планування:
  – Формування вимог до розробки.
  – Розробка критеріїв оцінки захищеності ПЗ.
  – Визначення специфікацій системного оточення.
  – Розробка формального проекту.
• Створення технічного завдання для розробки програмного забезпечення :
  – Ретельний аналіз інформаційної, функціональної та поведінкової моделі.
  – Декомпозиція робіт.
  – Вторинний аналіз ризиків.
• Розробка і тестування програмного забезпечення:
  – Розробка програмного забезпечення за допомогою безпечних методів програмування.
  – Створення проектної документації.
• Інсталяція програмного забезпечення:
  – Установка і впровадження продукту.
  – Тестування та аудит.
  – Створення документації для користувачів.
• Експлуатація та підтримка програмного забезпечення:
  – Незначна модифікація продукту.

ISSN 2663 - 4023

• Видалення продукту.

• Аналіз завершеного проекту.

Для підвищення ефективності оцінки захищеності розроблюваного ПЗ в першу чергу необхідно розробити методику, що дозволяє подолати недоліки підходу до оцінки захищеності на етапі проектування та розробки, недооцінку її ролі в забезпеченні необхідного рівня якості ПЗ, підміну тестування вразливостей процедурами типу перевірки працездатності на контрольному прикладі і т.п., а також використання методів безпечного програмування.

Щоб уникнути вразливості переповнення буфера, слід пам'ятати, що першою причиною цього є неправильна робота з зовнішніми даними та пам'яттю, а також відсутність захисту від підсистеми програмування та операційної системи.

Безперечно, усунути переповнення буфера доцільніше у першоджерелі – в вихідному коді. Однак, це вимагає навичок з управління пам'яттю і безпечної роботи з буфером. Найпростіший спосіб уникнути даної вразливості - просто використовувати мову, яка не дозволяє ручного управління буфером. Мова С має прямий доступ до пам'яті і має сильне типізування об'єктів. Однак, зміна мови програмування не завжди можлива. У цьому випадку доречно використовувати методи безпечного програмування, зокрема обробки буферів.

Іншим способом захисту є використання механізмів захисту, які надаються разом з деякими з сучасних операційних систем. За допомогою даних методів експлуатація вразливості переповнення буфера стає значно складнішим, однак повністю не виключає її. Практично для кожного методу захисту є методи обходження.

Щоб виявити і інші вразливості програмного забезпечення, скористайтеся технологіями аналізу програмного забезпечення на рівні коду. В цілому вони діляться на статичні і динамічні. Сфера застосування їх обох обмежена. Статичний аналіз дешевий і може працювати з великим обсягом коду, але його надійність становить 30-70%. Системи динамічного аналізу при знаходженні помилки відразу дозволяють отримати дані, на яких ця помилка проявляється (тобто не мають помилкових спрацьовувань), але застосування цих систем обмежене істотними вимогами до ресурсів. Тому найоптимальнішим виходом є комбінація обох методів.

Інші рекомендації, пов'язані з різними етапами розробки програмного забезпечення, наведені нижче:

1. Принципи забезпечення технологічної безпеки при обґрунтуванні, плануванні робіт і проектному аналізі ПЗ:
   a) Комплексність забезпечення безпеки ПЗ передбачає розгляд проблеми безпеки інформаційно-обчислювальних процесів з урахуванням всіх структур КС, можливих каналів витоку інформації і несанкціонованого доступу до неї, часу і умов їх виникнення, комплексного застосування організаційних і технічних заходів.
   b) Обґрунтованість засобів забезпечення безпеки ПЗ.
   c) Гнучкість управління захистом програм.

Важливо прийняти до уваги, що майже кожна платформа вразлива для переповнення буфера за такими помітними винятками: (1) J2EE - до тих пір, поки не будуть викликані нативні методи або системні виклики, (2) .NET - до тих пір, поки / небезпечний або некерований код не буде викликаний (наприклад, використання P / Invoke або COM Interop), (3) PHP - доки зовнішні програми та вразливі розширення PHP, написані на C або C ++, не можуть викликатись проблемами переповнення стека.

2. Принципи досягнення технологічної безпеки ПЗ в процесі його розробки:

a) Регламентація технологічних етапів розробки ПЗ, що включає впорядковані фази проміжного контролю, специфікацію програмних модулів і стандартизацію функцій і формату представлення даних.
b) Автоматизація засобів контролю керуючих і обчислювальних програм на наявність навмисних дефектів.
c) Централізоване управління базами даних проектів ПЗ і адміністрування технології їх розробки з жорстким розмежуванням функцій, обмеженням доступу відповідно до засобами діагностики, контролю і захисту.
d) Статистичний облік та ведення системних журналів про всі процеси розробки ПЗ з метою контролю технологічної безпеки.
e) Посилення обмежень щодо розміру буфера: необхідно валідувати всі вхідні дані.
f) Уникайте функцій, сприйнятливих до вразливості переповнення буфера.
g) Не використовуйте невідомі і незахищені бібліотечні файли. Вразливість, виявлена хакером у файлі бібліотеки, також буде існувати у всіх програмах, які використовують цей бібліотечний файл, що дасть хакерам ціль для потенційної атаки.

3. Принципи забезпечення технологічної безпеки на етапах випробувань та тестування:
a) Проведення випробувальних програм при екстремальних навантаженнях з імітацією активного впливу дефектів.
b) Проведення тестування продукту для виявлення дефектів, проведення статистичного та динамічного аналізу коду, тестування на проникнення.
c) Програмні продукти повинні бути сертифіковані відповідно до вимог безпеки.

4. Принципи забезпечення безпеки при експлуатації програмного забезпечення:
a) Забезпечення оновлення продукту під час їх експлуатації шляхом заміни окремих модулів без зміни загальної структури та зв'язків з іншими модулями.
b) Використовувані модулі, сервери та операційні системи повинні бути оновлені до останньої версії.
c) Статистичного аналізу інформації про всі процеси, робочі операції, відступи від режимів штатного функціонування ПЗ.

4. ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

По-перше, були проаналізовані і узагальнені загрози інформації в цілому. По-друге, для синтезу узагальненої моделі розробки безпечного програмного забезпечення висвітлювалась концепція життєвого циклу, проведено порівняння існуючих моделей життєвого циклу, кращих практик з розробки безпечного програмного забезпечення. По-третє, як приклад вразливості було продемонстровано переповнення буферу. Також описано механізми виявлення та захисту від вразливостей, які призводять до переповнення буфера. Однак важливо підкреслити, що для кожного способу захисту є методи, які дозволяють їх обійти.

Головним результатом роботи є рекомендації по створенню безпечного програмного забезпечення, що покращує існуючі підходи до розробки програмного

забезпечення та тестування з метою усунення вразливостей для нового програмного забезпечення та виявлення вразливостей для вже створеного.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] IEEE Standards Coordinating Committee, 1990. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos. CA: IEEE Computer Society, 169.
[2] Howard M., Lipner S. The security development lifecycle, vol. 8. – 2006.
[3] Futcher L., von Solms R. Guidelines for secure software development //Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology. – ACM, 2008. – С. 56-65.
[4] ДСТУ ISO/IEC/IEEE 12207:2019 (ISO/IEC/IEEE 12207:2017, IDT) Інженерія систем і програмних засобів. Процеси життєвого циклу програмних засобів
[5] ДСТУ ISO/IEC/IEEE 24765:2019 (ISO/IEC/IEEE 24765:2017, IDT) Інженерія систем і програмних засобів. Словник термінів
[6] База Common Weakness Enumeration [Електронний ресурс]. – Режим доступу: http://cwe.mitre.org.
[7] Spafford E. H. The Internet worm program: An analysis //ACM SIGCOMM Computer Communication Review. – 1989. – Т. 19. – №. 1. – С. 17-57.
[8] Hill M. D. et al. On the Spectre and Meltdown Processor Security Vulnerabilities //IEEE Micro. – 2019. – Т. 39. – №. 2. – С. 9-19.
[9] CVE-2019-0697 | Windows DHCP Client Remote Code Execution Vulnerability: https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-0697
[10] T. Babenko, S. Toliupa and Y. Kovalova, "LVQ models of DDOS attacks identification," *2018 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET)*, Lviv-Slavske, 2018, pp. 510-513.
[11] Intel I. and IA-32 architectures software developer's manual //Volume 3A: System Programming Guide, Part. – 64. – Т. 1. – №. 64. – С. 64.
[12] One A. Smashing the stack for fun and profit //Phrack magazine. – 1996. – Т. 7. – №. 49. – С. 14-16.
[13] Arutyun Avetisyan, Modern methods of static and dynamic analysis of programs for automation of processes for improving the quality of software: dissertation /Moscow. – 2012.
[14] Ivannikov V. P. et al. Static analyzer Svace for finding defects in a source program code //Programming and Computer Software. – 2014. – Т. 40. – №. 5. – С. 265-275.
[15] Brumley D. et al. Automatic patch-based exploit generation is possible: Techniques and implications //2008 IEEE Symposium on Security and Privacy (sp 2008). – IEEE, 2008. – С. 143-157.
[16] Avgerinos T. et al. AEG: Automatic exploit generation. – 2011.
[17] Bellard F. QEMU, a fast and portable dynamic translator //USENIX Annual Technical Conference, FREENIX Track. – 2005. – Т. 41. – С. 46.
[18] Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation //ACM Sigplan notices. – ACM, 2007. – Т. 42. – №. 6. – С. 89-100.
[19] Cadar C. et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs //OSDI. – 2008. – Т. 8. – С. 209-224..
[20] Avgerinos T. et al. Automatic exploit generation //Communications of the ACM. – 2014.

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 2 (6), 2019
ISSN 2663 - 4023

Viktoria Grechko
student Cybersecurity and Information Protection Department
Taras Shevchenko National University of Kyiv, Faculty of Information Technologies, Ukraine
ORCID 0000-0002-5190-4742
grechko.viktoria@gmail.com

Tetiana Babenko
Doctor of technical sciences, professor, professor of Cybersecurity and Information Protection Department
Taras Shevchenko National University of Kyiv, Faculty of Information Technologies, Ukraine
ORCID 0000-0003-1184-9483
babenkot@ua.fm

Larysa Myrutenko
Candidate of Technical Sciences, assistant professor of Cybersecurity and Information Protection Department
Taras Shevchenko Taras Shevchenko National University of Kyiv, Faculty of Information Technologies, Ukraine
ORCID 0000-0003-1686-261X
myrutenko.lara@gmail.com

SECURE SOFTWARE DEVELOPING RECOMMENDATIONS

Abstract. Adverse effects on information in the functioning computer systems of various purpose is carried out in order to violate their confidentiality, integrity and accessibility. These threats arise from software vulnerabilities and result in unauthorized access to data or leakage of sensitive information To solve this problem, firstly, an analysis of the software life cycle was carried out in order to determine the stages of software development. Secondly, taking into account the stages obtained, possible threats to information were identified. A buffer overflow vulnerability was considered as a basic example of a threat. Possible ways of exploiting this vulnerability are given, the pros and cons of detection and counteraction tools are analyzed. As a result, recommendations on the development of safe software are presented, both in general terms and more specific in order to avoid the buffer overflow vulnerability. Having using such recommendations, enterprises could reduce the risk of sensitive information breach and minimize outlane. The results obtained in the paper can also be used to make decisions about the possibility of operating the relevant software.

Keywords: secure software development; software development life cycle, vulnerabilities, buffer overflow, static and dynamic analysis, bufer overflow prevention mechanims.

REFERENCES

[1] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990, 1990. (in English).
[2] M. Howard, S. Lipner, “The security development lifecycle”, *Microsoft Press*, 2006. [Online]. Available: https://www.researchgate.net/publication/234792172_The_Security_Development_Lifecycle. [Accessed: 11- May - 2019]. (in English).
[3] L. Futcher and R. von Solms, "Guidelines for secure software development", Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries riding the wave of technology - SAICSIT '08, pp. 56-65, 2008. Available: 10.1145/1456659.1456667 [Accessed 11 May 2019]. (in English).
[4] DSTU ISO/IEC/IEEE 12207:2018 (ISO/IEC/IEEE 12207:2017, IDT) Systems and software engineering. Software life cycle processes, 2018. (in English).
[5] DSTU ISO/IEC/IEEE 24765:2018 (ISO/IEC/IEEE 24765:2017, IDT) Systems and software engineering. Vocabulary, 2018. (in English).
[6] "CWE - Common Weakness Enumeration", *Cwe.mitre.org*, 2019. [Online]. Available: https://cwe.mitre.org/index.html. [Accessed: 11- May- 2019]. (in English).
[7] E. Spafford, "The internet worm program: an analysis", *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 1, pp. 17-57, 1989. Available: 10.1145/66093.66095. (in English).

[8] M. Hill, J. Masters, P. Ranganathan, P. Turner and J. Hennessy, "On the Spectre and Meltdown Processor Security Vulnerabilities", *IEEE Micro*, vol. 39, no. 2, pp. 9-19, 2019. Available: 10.1109/mm.2019.2897677. (in English).
[9] "CVE-2019-0697 | Windows DHCP Client Remote Code Execution Vulnerability", *microsoft.com*, 2019. [Online]. Available: https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-0697. [Accessed: 11- May- 2019]. (in English).
[10] T. Babenko, S. Toliupa and Y. Kovalova, "LVQ models of DDOS attacks identification," *2018 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET)*, Lviv-Slavske, 2018, pp. 510-513. (in English).
[11] *Intel I. and IA-32 architectures software developer's manual*, Volume 3A: System Programming Guide, Part 1, September 2016. (in English).
[12] One, "Smashing the stack for fun and profit", *Phrack.org*, 1996. [Online]. Available: http://phrack.org/issues/49/14.html. [Accessed: 11- May - 2019]. (in English).
[13] Avetisyan, "Modern methods of static and dynamic analysis of programs for automation of processes for improving the quality of software", Doctor of Physical and Mathematical Sciences, Ivannikov Institute for System Programming of the RAS, 2012. (in English).
[14] V. Ivannikov et al., "Static analyzer Svace for finding of defects in program source code", *Proceedings of the Institute for System Programming of RAS*, vol. 26, no. 1, pp. 231-250, 2014. Available: 10.15514/ispras-2014-26(1)-7. (in English).
[15] D. Brumley, P. Poosankam, D. Song and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications", in *2008 IEEE Symposium on Security and Privacy*, IEEE Computer Society Washington, DC, USA, 2008, pp. 43-157. (in English).
[16] Avgerinos, T., Cha, S.K., Lim, B.T.H., and Brumley, D. "AEG: Automatic Exploit Generation," *Network and Distributed System Security Symposium, Internet Society*, San Diego, CA, 2011, pp. 283-300. (in English).
[17] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator", in *USENIX Annual Technical Conference*, Anaheim, CA, USA, 2005, pp. 41–46. (in English).
[18] N. Nethercote and J. Seward, "Valgrind", ACM SIGPLAN Notices, vol. 42, no. 6, pp. 89-100, 2007. Available: 10.1145/1273442.1250746 [Accessed 11 May 2019]. (in English).
[19] Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs", in OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008, pp. 209–224. (in English).
[20] T. Avgerinos, S. Cha, A. Rebert, E. Schwartz, M. Woo and D. Brumley, "Automatic exploit generation", Communications of the ACM, vol. 57, no. 2, pp. 74-84, 2014. Available: 10.1145/2560217.2560219 [Accessed 11 May 2019]. (in English).