|  |  |  |
| --- | --- | --- |
|  | **Софийски университет „Св. Кл. Охридски”**  Факултет по математика и информатика  *Катедра „Софтуерни технологии”* |  |

**ДИПЛОМНА РАБОТА**

на тема

„Предсказване на заплахи за сигурността на софтуерни системи”

Дипломант: **Димитър Петров Джонджоров**

Специалност: **Софтуерни технологии**

Факултетен номер: **M24463**

Научен ръководител:

**доц. д-р Александър Димов**

София, 2015 г.

**Съдържание** (препоръчителен обем без приложенията: от 60 до 80 стр.)

**Глава 1. Увод** (3-5стр.)

1.1. Актуалност на проблема и мотивация (0,5-1стр.)

1.2. Цел и задачи на дипломната работа (1-2стр.)

1.3. Очаквани ползи от реализацията (1-2стр.)

1.4. Структура на дипломната работа (0,5-1стр.)

**Глава 2. Преглед на предметната област (да се замени с конкретно заглавие според заданието)** (10-15стр.)

2.1. Основни дефиниции

2.2. Подходи, методи (евентуално модели и стандарти) за решаване на проблемите

2.3. Съществуващи решения (практически реализации)

2.4. Избор на критерии за сравнение и сравнителен анализ на решения/методи/стандарти/...

2.5. Изводи

**Глава 3. Използвани технологии, платформи и/или методологии (за практическото решаване на проблема)** (10-15стр.)

3.1. Изисквания към средствата (технологии, платформи и методологии)

3.2. Видове средства (технологии, платформи и методологии) и начин и място за използването им – сравненителен анализ

3.3. Избор на средствата (технологии, платформи и методологии)

3.4. Изводи

**Глава 4. Анализ** (10-15стр.)

4.1. Концептуален модел

4.2. Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...)

4.3. Качествени (нефункционални) изисквания (като напр. преносимост, използваемост, скалируемост, поддръжка, ...)

4.4. Работни (бизнес) процеси

4.5. Изводи

**Глава 5. Проектиране** (10-15стр.)

5.1. Обща архитектура – напр. слоеве, модули, блокове, компоненти...

5.2. Модел на данните (напр. база данни, файлова структура, ...)

5.3. Диаграми (на структура и поведение - по слоеве и модули, с извадки от кода)

5.4. Потребителски интерфейс (опционално)

5.5. Ресурсни и спомагателни модули (опционално)

**Глава 6. Реализация, тестване/експерименти и (евентуално) внедряване** (10-15стр.)

6.1. Реализация на модулите

6.2. Системна интеграция (опционално)

6.3. Планиране на тестването - тестови сценарии, процедури, ...

6.4. Модулно и системно тестване

6.5. Анализ на резултатите от тестването и начин на отразяването им

6.6. Експериментално внедряване (технологични изисквания, инсталиране, условия, използване, ...)

**Глава 7. Заключение** (1-2стр.)

7.1. Обобщение на изпълнението на началните цели

7.2. Насоки за бъдещо развитие и усъвършенстване

**Използвана литература** (min 10 литературни източника – статии, книги, с, форматирани съгласно MLA Style - <http://www.library.mun.ca/guides/howto/mla.php>)

**Приложения (опционално)**

**Приложение 1: Терминологичен речник (опционално)** (1-2стр.)

**Приложение 2, 3, ....:** напр. код или извадки от кода, наръчник на потребителя, екрани от потребителския интерфейс, спецификации, диаграми, и др.

1. Увод
   1. Актуалност на проблема и мотивация

В днешния зависим от технологиите свят, въпросът за сигурността и личното пространство е все по-належаш и проблемен. Почти всеки един аспект на нашето ежедневие и работа е зависим от технологиите и взаимодействието със сървъри. Сървър представлява машина(физическа или виртуална), чиято основна дейност е да съхранява и осигурява достъп до съответната инфомация. Дотогава цялата информация за даден архив е била трудна за поддържане на повече от един компютър. Това е така защото всяка една промяна в информацията, която тази система използва изиска преноса на тази информация ръчно до другите Тези сървъри могат да предоставят най-различни услуги като започнем от системите, които използваме на работните си места, минем през електронни пощи и социални мрежи и достигнем до системи, за които повечето от нас не подозират, че използваме, като например тези в градския транспорт, светофари и други. Макар и да не осъзнаваме, около нас постоянно има движещи се потоци от данни.

Именно наличието на тези обеми от данни спомага за това Интернет да се превърне в основно средство за комуникация и обмен на информация в наши дни. Едно от основните му предимства е това, че тези услуги са публично достъпни от всяко едно устройство. Това прави много по-удобна работата с дадена програма или приложение, защото работата с него не ни обвързва с конкретна машина.

Важна роля има и популяризирането на смартфоните. През 90-те години е рядко срещано някой да притежава мобилен телефон поради това, че не са достатъчно достъпни. През годините от 2007 до 2010 се наблюдававъвеждането на смартфоните с touch screen дисплей, при които вече не сме ограничени от типа клавиатура, а и екрана е по-голям и позволява разглеждането на повече съдържание. Така стигаме до наши дни и ежедневно милиарди хора използват смартфоните за работа с Интернет.

В наши дни също така и много институции съхраняват своите данни на сървъри, които са достъпни през мрежата. Именно поради популярността на последната, не са рядкост и злоупотребите, които се извършват през нея. Това е така, защото данните, които преминават през нея могат да бъдат проследени и достъпени от трети лица. Заради такива проблеми в последните 20 години възниква и един нов клон на софтуерната разработка – осигуряване на софтуерната сигурност.

* 1. Цел и задачи надипломната работа

Основна цел на тази дипломна работа е да се реализира алгоритъм за предсказване на атаки по сигурността на софтуерни системи чрез откриване на аномалии, чието основно предимство е оптимизиране на времето за проверка на заявките за съдържание на зловреден код. Задачите, произлизащи от тази цел, са най-общо обзор и анализ на проблемната област, дефиниране на модел за предсказване на атаки и изисквания към софтуерен инструмент, който да го имплементира. След това ще продължим с реализация на инструмента, тестване и експерименти. Нека сега да структурираме гореспоменатите задачи::

1. Обзор и анализ на проблемната област – видове атаки, методи и т.н.
2. Дефиниране на модел за предсказване на атаки чрез анализ на аномалии в потребителските заявки.
3. Дефиниране на изискванията към софтуерен инструмент за предсказване на атаки чрез аномалии.
4. Проектиране на софтуерния инструмент.
5. Реализация.
6. Тестване и експерименти.
   1. Очаквани резултати от реализацията

С успешната реализация на тази дипломна работа предоставяме документ, анализиращ проблема, свързан със сигурността на софтуерните приложения и в частност с атаките от тип SQL инжекция. В него ще е наличен и анализ на съществуващи методологии и реални практически решения и ще бъде предложен алгоритъм за справяне с проблема на базата на оптимизация на заявките и операциите върху заявката с цел проверка за индикатори за атака от този вид. Освен документа ще е налично и уеб приложение, илюстриращо горния алгоритъм на практика. Всяка организация, която цели справяне с проблема, ще може да използва документа за придобиване на начална представа за обхвата на проблема и съществуващите решения, а приложението – за директно онагледяване на предоставеното решение, без да влага излишни средства в собствено проучване.

* 1. Структура на дипломната работа

Структурата на текущата дипломна работа е постепенно да въведе читателя в проблемната област доката не стигне до конкретния проблем и неговите решения. Целта на това е той да бъде запознат с контекста на атаките от тип SQL инжекция, за да може по-добре да установи причините за това те да са най-разпространените такива и все още да не е налично решение за тях. Налично е кратко въведение в проблемната област на уеб приложенията и тяхната сигурност. Следва кратък анализ на най-разпространените атаки, преди да стигнем до задълбочено разглеждане на темата и видовете атаки от тип SQL инжекция и техни решения под формата на методологии и алгоритми. При наличен обзор на проблемната област, е разработена методология, целяща да предложи защита от основните атаки от разглеждания тип при оптимизиран брой операции и итерации. Накрая е разгледано допълнително разработеното приложение, илюстриращо на пректика вече описаната методология и следващи стъпки за подобряване то му.

Нека представим структурата на настоящата дипломна работа. Във втора глава ще въведем читателите в областта на софтуерната сигурност и в частност атаките към нея(секция 2.1.2.). След това ще разгледаме основните типове атаки(секция 2.1.3) и ще се концентрираме върху различните видове атаки от тип SQL инжекция(секция 2.2). В глава 2.3. ще анализираме различните архитектурни подходи за справяне и смегчаване на последствията от евентуални атаки от тип SQL инжекция.В рамките на секция 2 ще се запознаем и двата реализирани алгоритъма за предотвратяване на SQL инжекции(глава 2.4).

Следва 3та секция, в която ше разгледаме алгоритъма, който ще предложим за откриване на аномалии в потребителското поведение от вида сляпа SQL инжекция и ще го сравним с вече разгледаните алгоритми от секция 2.4. След това ще представим и технологиите в секция 4, които сме използвали при реализацията на инструмента, скойто ще илюстрираме алгоритъма в предишната секция. Освен технологиите, ще представим и различни диаграми, свързани с приложението – на базата данни, архитектурна и потребителски сценарии. Тях ще разгледаме в секция 5. Предпоследната глава 6 ще е концентрирана върху тестовият инструмент, който сме създали като допълнение към дипломната работа. С него целим да покажем действието на нашия алгоритъм в почти реални условия, в които много потребители с различни IP адреси изпращат заявки към сървъра, на който се намира приложението. Ще завършим дипломната работа с глава посветена на заключения за текущата дипломна работа, както и насоки за бъдещо развитие.

1. Софтуерна сигурност и атаки към нея
2. Въведение в областта

Именно поради това, че популярността на Интернет расте и ще продължава да го прави все повече, расте и важността на въпроса касаещ сигурността на тези приложения, намиращи се на сървъри и информацията прехвърляна през мрежата. Именно това, че Интернет вече е достъпен до изключително много хора, расте и броят на злоупотребите с него. Първите атаки над даден софтуер са извършени през 70те, когато се появяват така наречените хакерски банди, чиято сновна цел е осъществяването на безплатни телефонни разговори. 90те години се считат за зората на съвременната индустрия за осигуряване на софтуерна сигурност. Това е и десетилетието, в което се появява и един от най-популярните видове софтуерни атаки – тези за отказване на достъп. [25]

Макар и да се наблюдава намаляване на броя атаки последните години, това се дължи по-скоро на тенденцията атаките да стават по-сложни и мащабни, защото вече има разработени системи за превенция на част от съществуващите атаки. Евентуална причина софтуерните атаки да стават все по сложни е значително улеснения процес на разработка на по-сложен и комплексен софтуер.

1. Видове уеб приложения

В зората на Интернет, докато той все още не е бил толкова широко разпространен световната мрежа не е била толкова развита, начинът за представяне на даден тип информация в браузърите са били уеб сайтовете..

Основният въпрос е как потребителят ще ще взаимодейства със сайта, за да получи информацията, която му е необходима. След като даден потребител достъпи до началната страница на даден сайт, той може да достигне до някоя от другите страници по различни начини, най-разпространения от които е чрез даден линк, който просто кара браузъра да изпрати заявка и да визуализира друга страница. Потребителят може просто да въведе адресът на търсената страница и тя ще се зареди, но за тази цел той трябва да познава файловата структура на уеб сайта. Файловете, свързани с даден уеб сайт се намират на даден сървър и при заявка за дадена страница, сървърът връща съответната.

В началото сайтовете са по-малки и не толкова развити и информативни. Постепенно обаче те започват да стават все по-обемни и сложни като структура, както и да се увеличава трафика към сървъра, който съхранява страниците. Появява се необходимостта даден сайт да се пази на повече от един сървър с цел справяне с тези условия.

Постепенно, поради нарастващото потребление на Интернет в световен мащаб, започват и да се задават по високи стандарти за това как да изглежда даден сайт и какви функционалности да предлага. Възниква необходимостта от по-комплексни сайтове и така се появяват първите уеб приложения.

Уеб приложение наричаме съвкупност от различни уеб и браузърни технологии с цел създаване на софтуер, чиято цел е решаването на една или повече задачи. Характерни за тях са т.нар. персонализиране на уеб съдържание, изпълним код в браузъра на потребителя и други. Първото представлява отчитане на потребителя, който взаимодейства с приложението на определения браузър, и показва уникално за всеки потребител съдържание във визуализираната от браузъра страница. Това може да бъде потребителска мета-информация и други. Типични примери са електронните магазини, онлайн поща и всеки един модул за идентифициране на потребител на дадено приложение. Втория аспект представлява едно ниво по-високо в развитието на структурата на съдържанието на дадена страница, а именно вече дадени потребителски действия могат да предизвикат изпълнението на даден код, което би довели до промяна на съдържанието на страницата. По този начин последната вече не е чисто статична като при стандартните браузъри, а вече е динамична.[9] Най-известният и използван език за обработка на динамично съдържание в браузъра е JavaScript. Той представлява скриптов език за създаване на динамични уеб страници.Скриптов език наричаме такъв, чрез който се създават скриптове или програми, които вместо да се компилират, се интерпретират директно. Понеже уеб приложенията, както и уеб сайтовете се съхраняват на сървъри. Затова процесът на инсталиране на дадено приложение в сървърна среда наричаме вграждаме на приложение.

Положителните страни на уеб приложенията са, че те не трябва да се инсталират на устройството на даден клиент, за да може да работи с него той трябва просто да го достъпи чрез даден браузър. Това спестява много работа, свързана с инсталирането на софтуера много пъти. В наши дни повечето големи приложения на Майкрософт и други корпорации имат създадени уеб версии. Това улеснява много тези организации, защото иначе те не само трябва да произведат огромно количество хардуерни носители(най-вече дискове), на които да се намират инсталационните файлове, но и да следят за възникващи проблеми на всяко едно от устройствата, на които е инсталиран софтуера. От друга страна дадено уеб приложение е много по-лесно за поддръжка и поправка на дадени проблеми с него. Също така при евентуално издаване на нова версия или на някакъв ъпдейт, трябва отново да се следи за това на всички компютри да се инсталира новата версия, докато в нашия случай усилията са концентрирани само върху заместването на старата версия на приложението на сървъра с нова.

Разбира се както всичко останало и уеб приложенията имат своите недостатъци. Един от тях е засиленият поток към сървъра, на който е разположено приложението. Трябва много добре да е проектирана сървърната архитектура, така че да не се получи претоварване на сървърът/те. Също така е добре да се предвиди и стратегия ако броят на тези потребители нарасне още повече.

1. Софтуерна сигурност

Сигурност на дадено софтуерно приложение представлява набор от добри практики или идеи, които след подходяща имплементация го защитават от злонамерени атаки, но така, че самото то да функционира нормално и според изискаванията. Казваме, че една система е сигурна, ако тя притежава качествата цялост, поверителност и достъпност.

Цялост на дадена система представлява поддръжката на постоянство, точност и достоверност на данните през целия и жизнен цикъл. Това включва и подсигуряване на това информацията между клиента и сървъра да не може да бъде променяна от неоторизирани лица. Добри практики за запазване на целостта на дадена система са въвеждането на механизми за криптиране на данните, които се обменят между клиентите и сървъра и използване на резервни копия на данните, които могат да бъдат загубени при евентуална атака, която да разруши тази характеристика на системите.

Поверителност на дадено софтуерно приложение представляват мерките, които трябва да се вземат по време на разработването на приложението така, че да не се стигне до изпращане на съдържание на грешния човек, като същевременно да се подсигури получаването му от правилния потребител. Евентуален пропуск в разработване на механизмите за софтуерна сигурност на дадено приложение би довело до неспазване на това условие и което е по-лошо – до попадане на информация за конкретен потребител в недоброжелателни ръце. Добри практики за предотвратяването на такива сценарии е използване на криптиране на данните за потребителски акаунти, въвеждане на процедура за вписване в системата по поне два фактора(най-често потребителско име и парола).

Достъпност представлява качеството на системата да бъде достъпна за използване от потребителите, които имат определените права за това, по всяко време и на всяко устройство. Едни от най-популярните атаки в наши дни са атаките за отказ на услуги, чиято цел е именно да открият слабост в тоза звено на софтуерните системи. Евентуални последствия от една такава атака биха били невъзможност за достъп до приложението или некоректно работеща система. Това от своя страна би довело до големи загуби за съответните организации, които използват приложението, както и за тази, която го притежава. Добри практики за справяне с този проблем са допулнителни механизми за достъпност на системата като клъстери, за справяне с евентуални сривове в нея добра честота на комуникацията между клиентите и сървъра.

1. Видове атаки

С увеличаване на популярността на Интернет и Световният уеб, постепенно расте и броят на злоупотребите в него. Особено в последните години се е увеличил и броят на различните видове софтурени атаки. Само за периода от 2006 до 2014 броят атаки годишно се е увеличил с над 2000. Сред най-популярнитесред тях са SQL инжекцията, разбиване на акаунти, кръстосано скриптиране и други. За 2014 година тези три вида атаки запазват своите челни позиции, като според Forbes се наблюдава нарастваащ брой на атаките в глобален мащаб. Лошата тенденция, която те отбелязват е, че извършителите на тези кибер предстъпления се усъвършенстват до такава степен, че информацията, която бива открадната достига до черния пазар в рамките на няколко дни. Също така тенденцията е към нарастване на броя на определени атаки, като атака за отказване на достъп, чиито брой се е удвоил в рамките на последната година. Сред най-големите атаки на 2014 година са открадване на персонална информация за 233 милиона потербители от Ebay, открадване на информация за кредитни и дебитни карти от системата на веригата ресторанти P.F. Chang’s и кражбата на 600000 потребителски записи на клиенти на пицарии Domino’s, за която хакерската организация Rex Mundi е поискала 40000 американски долара. Въпреки нарасващият брой на атаки от други типове, инжекцията на код и по-конкретно тази от тип SQL запазва стабилно първата си позиция. [16][17] Според проучване на OWASP за 2013 година това е най-разпространения тип атака, следван от „повредена” автентикация и управление на сесиите, междусайтово скриптиране и несигурна директна референция.[15]. Нека все пак да разгледаме накратко и другите три типа преди да преминем към по-подробен анализ на основната тема на тази дипломна работа.

Междусайтово скриптиране(или Cross-Site Scripting (XSS)) представлява атака, която се основава на липсата на контрол над съдържанието, подавано от потребителя. В такъв случай атакуващият може да модифицира информацията чрез изпълняване на скриптове в клиентската част на приложението и така да доведе до получаване на поверителна информация за даден потребител като сесиини идентификатори или пренасочване на потребителя към злонамерени сайтове. Агент на заплаха считаме всеки потребител, който може да изпрати ненадеждни данни към системата. При успех на самата атака злонамерениият потребител би се сдобил с потребителски сесии, препрати към други сайтове, вкара злонамерено съдържание, заразяване на браузъра с мейлуер и други. Последствията включват кражба на поверителна информация за клиентите на дадена система и бизнес такива от публичничността на такава слабост.

Нека да разгледаме един пример за този вид атака. Нека разгледаме приложение, което динамично генерира изходния HTML код, визуализиращ се в браузъра, без специални проверки, например:



По този начин атакуващият може да промени параметъра СС в браузъра например със следното:



По този начин се изпраща сесииния идентификатор на съответния потребител към сайта на атакуващия.

„Повредена” автентикация и управление на сесиите(Broken Authentication and Session Management) включва компроментиране на пароли, ключове, сесии или да се използват други недостатъци в имплементацията на модулите за управление на аутентикацията и сесиите. Източници на атаката могат да бъдат всички - и регистрирани потребители или такива, които не са. Целта на атакуващия е да достигне до поверителна информация за даден потребител на системата. Така той може да използва присвоената самоличност за да извърши недобронамерени действия спрямо системата. Понякога обхвата на тази атака може да бъде дори присвояване на информация за всички потребители на системата. Поради факта, че можеш да използваш системата като някой друг, основен обект на атаките са акаунти със специални привилегии. Последствията могат да варират от бизнес такива от самото откриване на въпросната слабост до кражба на информация и евентуално нарушаване на правилните функции на системата.

Пример за такъв тип атака е некоректно конфигуриране на сесиите. В такъв случай, когато даден потребител на съответната система е влязъл в нея от публичен компютър, то съществува рискът той вместо да се отпише просто да затвори прозореца, в който е използвал приложението. По този начин евентуалният атакуваш може да седне на компютъра, да отвори приложението в същият браузър и да продължи да използва сесията на предишния потребител.

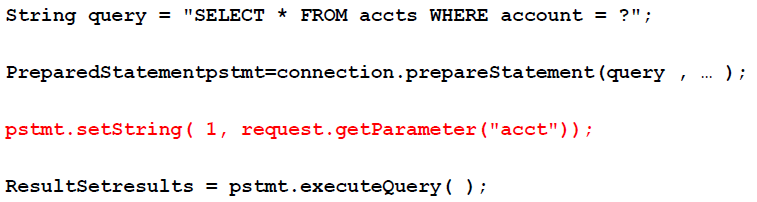
Друг такъв пример е приложение което поддържа презаписване на сесиините идентификатори и за целта ги поставя в URL на заявката, например:



Така ако атакуващият достигне до това URL, той ще може да използва сесията на потребителя и по този начин и негови данни като номер на кредитната му карта и други.

Несигурна директна референция (Insecure Direct Object Reference) към обект се нарежда в проучването на OWASP веднага след предишните три. Тази атака представлява достъпване до имплементационен ресурс, като файл, директория или ключ за база данни например, който би трябвало да не е публичен, но е поради слабости в разработката на системата. Оттук нататък всеки атакуващ може да промени тези вътрешни връзки към обекти, за да получат достъп до данни, за които нямат права. Атакуващият трябва да е оторизиран потребител на системата, за да може да извърши такава атака. Обикновено атаката е свързана с променяне на ясния път до даден системен ресурс с друг такъв, за който потребителя няма права. Ако приложението е разработено така, че да не допуска достъп до атакувания ресурс от потребители без необходимите права, такава атака няма да бъде успешна. В противен случай недоброжелателят ще получи достъп до ресурс, за който няма права. Така може да се получи компроментиране на всички данни, коитомогат да бъдат достъпени от дадената променлива, съхраняваща съответния път. И тук последиците са основно бизнес насочени и свързани с присвояване на поверителна информация и евентуално разпространяване на въпросната слабост може да повлияе на имиджа на организацията.

Нека разгледаме един пример. Нека имаме следната конструкция в кода:



В случая номерът на съответния акаунт се взема директно от параметъра на заявката. По този начин атакуващия просто трябва да промени стойността на параметъра acct с произвлен номер на акаунт и ще получи информация за него.

1. Атаки от тип SQL инжекция

В основата си този тип атака представлява вмъкване на определени SQL команди с цел да модифицира изпълнението на съответната заявка на сървъра, на който е разположена съответната база данни. В зависимост от степента на защитеност на системата или приложението, атакуващият може да извлече или промени поверителна информация, да повлияе върху самото изпълнение на заявката или дори да изтрие информация от нея или дори самата база. Както се предполага от името, този вид атака принадлежи към групата на атаки от тип инжекция поради факта, че в информацията, използвана за съставяне на заявката към сървъра се инжектират SQL команди.[12]

Основната цел на този тип атаки е извършителят да получи достъп до информация за дадено лице или организация, съхранявана на сървъра, като главната подбуда е злоупотреба с нея. Атакуващият, обаче, може да използва SQL инжекция, за да унищожи данни от съответната база данни.

Успешното изпълнение на злоупотреба от тип SQL Инжекция евентуално би довело до следните проблеми:

* Загуба на поверителност на информацията, която се съхранява на сървъра поради факта, че външен човек разполага с нея.
* Несигурност на потребителските акаунти в системата, тъй като атакуващият може да се впише в нея като произволен потребител. В повечето случай това касае акаунтите с администраторски права, защото те биха били основната цел на всеки злонамерен потребител.
* Загуба на целостта на информацията, тъй като недоброжелателят може да промени или унищожи ценна поверитена информация от базата данни.

Макар и да няма официална класификация на атаките от тип SQL инжекция, ние можем да ги разделим на три вида: слепи, за извличане на информация и деструктивни, като в следващите секции ще разгледаме по-подробно всеки един от тях.

1. Сляпа SQL инжекция

Това представлява атака, която базирайки се на твърдения от булев тип, цели да анализира отговора на приложението и получи отговор на определени въпроси, касаещи базата данни и сървъра, като например версията му или имената на определени таблици. Този тип злоупотреби се извършват от даден атакуващ преди да извърши атака, с която да извлече информация или да модифицира или изтрие такава. Основната прочина затова е, че тогава той няма представа за структурата на базата данни, таблиците в нея, както и самия сървър. Също така този приом се използва, когато приложението е конфигурирано така, че да показва съобщения получени директно от базата данни.

Нека сега да разгледаме примери за слепи атаки от тип SQL инжекция като анализираме следната ситуация:

Нека предположим, че имаме база данни с име bank\_system и таблица user\_data в нея, чиито идентификатор е колона с име ID. От инерфейса на системата се изпраща заявка за получаване на информация от съответната таблица към следния URL:

http://website.org/userData.php?id=2 (1)

След получаването на съответната заявка(1) се конструира следната заявка(2) към базата данни.

SELECT \* FROM bank\_system. user\_data WHERE ID = 2 (2)

Целта на атакуващият е да инжектира твърдение, което със сигурност да върне отрицателен отговор при изпълнение. Това се постига като се симулира заявка(3) със следния URL

http://website.org/userData.php?id=2 AND 2=3 (3)

Това би довело до съставянето на следната заявка(4)

SELECT \* FROM bank\_system. user\_data WHERE ID = 2 AND 2=3 (4)

Ако системата не е защитена, то резултатът от изпълнението на последната трябва да е празна колекция от записи на съответната таблица. Това е и целта на атакуващия – да установи дали системата е уязвима на атаки от тип SQL Injection. За да бъде сигурен обаче, той симулира и заявка, която винаги трябва да връща положителен резултат при изпълнение, например:

http://website.org/userData.php?id=2 and 1=1 (5)

След това той просто трябва да сравни резултатите от двете заявки и ако те се различават, то със сигурност системата е уязвима.

Следващият тип сляпа атака е времевата. Тя се казва така, защото използва механизми за забавяне на изпълнението на заявката и по този начин отговаря на въпроси, зададени от атакуващия, например какъв е първият символ на паролата на даден потребител. Нека онагледим това с пример като разгледаме отново същия сценарии от преди малко. Нека този път атакуващият да знае името на базата и таблицата, както и името на колоната, пазеща потребителската парола. Типична заявка от този тип е

http://website.org/userData.php?id=2 UNION SELECT IF(SUBSTRING(user\_password,1,1) = CHAR(50),BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),null) FROM users\_data WHERE ID = 1; (6)

Това IF твърдение в заявка (6) представлява проверка дали първият символ на потребителската парола е ‘2’(CHAR(50) е кодът на числото 2). В случай, че това е вярно твърдението BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')) ще изпълни функцията ENCODE 5000000 пъти и ще забави изпълнението на заявката. По този начин атакуващият ще разбере, че твърдението, което допуска наистина е вярно. По този начин той може да продължи, докато не получи цялата му нужна информация. В случая, в който имаме MySQL база данни, ние можем да извлечем имената на таблиците в нея и по друг начин, а именно

http://website.org/userData.php?id=2 UNION SELECT table\_schema, table\_name, 1 FROM information\_schema.tables; (7)

Тук в заявка (7) единствената трудност е да се уцели броят на колоните в резултата, но и това е възможно просто чрез прибавяне или премахване на 1ци във втория SELECT израз.

Последният тип, който ще разгледаме е сляпа атака с цел разпознаване на типа база данни, която използва приложението. Един типичен подход за това е да изплозваме функциите за намиране на настоящата дата и време. Ще се върнем отново към горния пример, но този път потребителят, който се готви да я атакува не знае какъв е типът на базата данни. Затова той може да изпрати заявка от типа (8):

http://website.org/userData.php?id=now() (8)

По този начин може да установи дали става въпрос за MySQL сървър или за MSSQL сървър. Това ще стане тъй като now() е валидна функция за първия тип и изпълнението на последващата заявка

SELECT \* FROM bank\_system.user\_data WHERE ID = now() (9)

Ако изпълним заявка (9) в MSSQL сървър би довело до грешка, която ако системата не е защитена ще стигне до атакуващия и той ще разбере това.[13]

1. SQL инжекция с цел извличане на информация

След като събере мета информация за базата данни и таблиците в нея, най- често срещаният случай е атакуващия да иска да извлече информация за базата данни. В този случай както вече казахме най-уязвими са акаунтите с администраторски права, защото чрез тях атакуващия може да влезе в системата и да отвори модули от нея, достъпни само за потребители с администраторски права. Нека отново разгледаме няколко примера в контекста на горния сценарии. Нека в следващият случай да става въпрос за форма за вписване, в която има две полета – за потребителско име и парола. При въвеждане на стойности в двете полета, интерфейсната логика изпраща заявка(10) със следния най-общ вид:

http://login.website.org/userData.php?username=dimitar&password=samplePassword (10)

от която се конструира SQL заявка(11) от типа

SELECT \* FROM bank\_system.user\_data WHERE username=dimitar AND password=samplePassword (11)

и при установяване на съществуващ такъв запис в съответната таблица, вписва потребителя и го идентифицира. Атакуващият тук се опитва да влезе като даден потребител, като както казахме вече, в повечето случай става въпрос за администраторски профил. Затова то тръгва от презюмцията, че съответния акаунт е с потребителко име admin. Един вариант да се идентифицира като този потребител е като симулира следната заявка(12):

http://login.website.org/userData.php?username=admin’ -- &password= (12)

Така ако системата не е защитена от атаки от тип SQL инжекция, ще конструира следната заявка(13) към сървъра, на който се намира базата данни:

SELECT \* FROM bank\_system. user\_data WHERE username=’admin’ -- AND password= (13)

Тъй като -- ще се интерпретира като закоментиране на останалата част от заявката след нея, резултатът ще се състои от всички записи, за които потребителското име е admin. Така недоброжелателят ще заблуди системата и тя ще го впише като съответния потребител. В следващия пример ще разгледаме случая, в който атакуващият вече се е вписал в системата в даден профил(с администраторски права или без) и се опитва да извлече цялата информация от дадена таблица. Ако системата връща колекция от записи на съответната таблица, чрез заявка (1) от по-горния тип, то може да се симулира заявка(5) от вече разгледания от нас тип, която при успешно изпълнение ще върне всички записи от съответната таблица.

Последният пример за заявка от този тип, който ще разгледаме използва UNION конструкция. Атакуващият изпраща заявка(16) от типа:

http://website.org/userData.php?id=2 UNION SELECT col1, col2,… coln FROM bank\_system.user\_data (16)

По този начин ще се конструира друга такава(17) от типа:

SELECT \* FROM bank\_system. user\_data WHERE ID = 2 UNION SELECT col1, col2,… coln FROM bank\_system.user\_data; (17)

Това твърдение отново ще върне всички записи от съответната таблица. Тук обаче атакуващият трабва да знае името на базата и таблицата, както и списъка с колони от първото SELECT твърдение. Първите две той може да достъпи по познатия вече начин от предишните примери, а последният може да бъде разгледан от структурата на върнатия резултат. В него ако разработчиците не са взели специални предпазни мерки имената на полетата ще отговарят на наименованията на съответните колони от таблицата.

1. Деструктивни атаки от тип SQL инжекция

Двата досегашни типа атаки са опасни за системата или по-скоро за поверителността на информацията в нея, но безспорно третия тип е най-критичен като последствия за съдържанието и целостта на информацията на сървъра, на който се намира базата данни. Това е поради факта, че успешна атака от този тип може да доведе до промяна на част от данните в базата, а често и до унищожаването на цели таблици и бази от сървъра. Лошото в този случай не се състои само в това, че се изтрива поверителна информация, а в това, че в повечето случаи преди това атакуващият вече е успял да я извлече. Така, ако не сме предприели специални мерки, тези данни не само са откраднати, но вече не съществуват на нашия сървър и е трудно да бъдат възстановени.

Както вече споменахме този вид атаки могат да бъдат разделени на два типа, като и за двата отново анализираме вече познатия сценарий:

Първият са тези, които модифицират съдържанието на дадена таблица. При него изкуствено се добавя съдържание в заявката към сървъра, чиято цел е да промени стойността на една или повече колони на дадени записи от нея. Отново разглеждаме заявката (1).

Използвайки нея и знаейки името на базата данни и таблицата – bank\_system.user\_data, както и това на съответната колона, която иска да промени, например password, атакуващият може да симулира следната заявка(18):

http://website.org/userData.php?id=2; UPDATE bank\_system.user\_data SET password = ‘1111’ WHERE username = ‘admin’; (18)

Това ще доведе до изпълнението на следната SQL заявка(19):

SELECT \* FROM bank\_system.user\_data WHERE ID = 2; UPDATE bank\_system.user\_data SET password = ‘1111’ WHERE username = ‘admin’; (19)

Така интерпретаторът ще изпълни освен първата заявка и ще бъде инструктиран да изпълни и втората, чиято цел е да промени паролата на профил с потребителско име admin. Така съответния недоброжелател може вендага да се впише като администратор и да извърши и други злоупотреби.

Вторият вид касае атаки, изтриващи информация на сървъра. При тях недоброжелателят „инжектира“ SQL код, чиято цел е да изтрие или таблица или цяла база данни. Отново имаме същата ситуация, но този път атакуващият ще изпрати следната заявка(20):

http://website.org/userData.php?id=2; DROP TABLE bank\_system.user\_data; (20)

След това ще се изпълни друга такава(21):

SELECT \* FROM bank\_system.user\_data WHERE ID = 2; DROP TABLE bank\_system.user\_data; (21)

Второто твърдение ще доведе до изтриването на таблицата bank\_system.user\_data от паметта на сървъра. Със същия успех атакуващия може да изтрие и цялата база данни просто като симулира подобна заявка(22):

http://website.org/userData.php?id=2; DROP DATABASE bank\_system; (22)

Последващата такава ще доведе до премахване на цялата база данни.

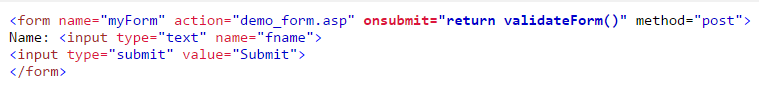
1. Подходи и методи за решаване и облекчаване на последствията от атаките от тип SQL инжекция

Най-общо можем да разделим подходите за предотвратяване на атаки от тип SQL инжекция на три вида: архитектурни, алгоритмични и такива за смегчаване на последствията. Сред първите са валидация на потребителските полета, използване на акаунт за базата данни с по-малко права и използване на системни запазени процедури. Към втория тип спадат предварително приготвените заявки, а към последния – създаването на резрвно копие на базата данни. Нека сега да разгледаме поотделно всеки един от тези подходи.

1. Валидация на полетата във дадена форма

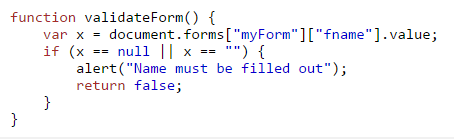
Валидация на полетата в дадена форма е процес на осигуряване на това, че данните, с които работи дадена програма(приложение) са коректни и използваеми. Тя представлява само по себе си защитна стена за нашата апликация. Всеки един пакет от инструменти за създаване на интерфейс включва механизъм за реализация на валидиране на съдържанието на полетата на дадена форма. Това се постига по различни начини в различните езици.

Нека вземем предвид Javascript – най-популярния език за разработка на динамично съдържание в клиентската част на всяко уеб приложение. При него постигаме валидация чрез контролен метод, който се извиква при натискане на бутона за изпращане на формата към сървъра.



*Фигура 2.3.1.1.*

В този случай, представен на фигура 2.3.1.1. когато натиснем въпросния бутон първо ще се изпълни метода validateForm, който има следния вид(фигура 2.3.1.2.)



*Фигура 2.3.1.2.*

В този пример формата има едно поле, чието име е fname. В самия метод ние вземаме стойността на самото поле, като първо вземаме стойностите на всички полета във формата(чрез document.forms[“myform”]) и след това достигаме до търсената такава посредством името и.

Съществуват и други методи за валидация на съдържанието на текстовите полета, като например използването на предварително написани библиотеки като Verify.js [3].

Нека сега да разгледаме валидацията при основния език за съдържание в браузъра, а именно HTML. При него също има начини за постигане на контрол над съдържаниено на дадено поле. Първият от тях е чрез използване на required атрибут в съответния HTML таг:



Чрез него задължаваме потребителят да попълни полето преди формата да бъде качена за обработка на сървъра.

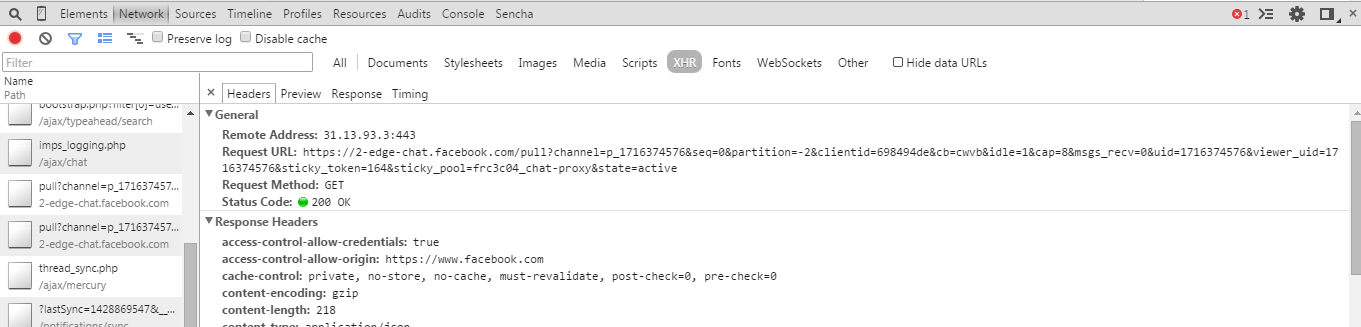
Безспорно обаче най-важният приом за валидация е използването на атрибута type за input тага на съответното поле. Има няколко стойности, които този атрибут може да приема – text, email, url, number, tel, date, отговарящи съответно на текст, имейл, хиперлинк, число, телефон и дата. За да валидираме съдържанието на дадено поле просто трябва да прибавим съответния атрибут type и желаната стойност, както е илюстрирано в следния пример на фигура 2.3.1.3.:



*Фигура* *2.3.1.3.*

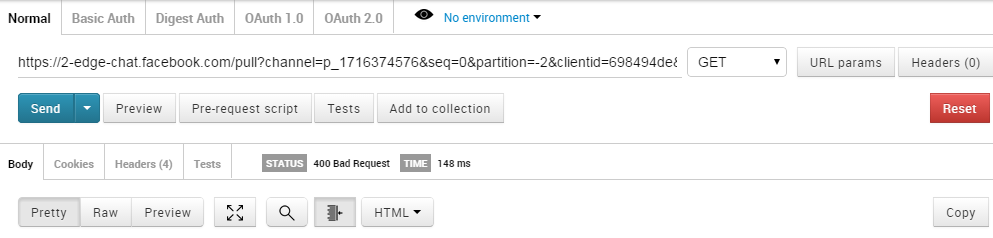
Тук сме избрали полето да се валидира като имейл, тоест за да бъде одобрена за изпращане формата трябва задължително(required) да сме попълнили полето за имейл със съдържание от вида [a-zA-z\_]@[a-z].[a-z](общият вид на регулярния израз за валидиране на имейл). С въвеждането на HTML5 се въвеждат и други допустими стойности за атрибута type, а именно color, date, datetime, datetime-local, month, search, time, week.[4]

Въпреки че валидацията на полетата на всяка една онлайн форма са полезни и даже необходими, те не са достатъчни за да се предпазим от атака тип SQL Инжекция. Причината за това е, че тя ни предпазва от недоброжелателна потребителска дейност, но само в рамките на съдържанието въведено в дадено поле от формата. Проблемът идва от това, че даден злонамерен потребител има възможността да проследи съдържанието на тялото и заглавната част на заявката, чрез която формата се изпраща до сървъра, използвайки инструменти като Chrome Web Inspector и други. Например, мога да отворя и прегледам заглавната част на следната заявка от фигура 2.3.1.4.:



*Фигура 2.3.1.4. Общ вид на заглавната част на разглежданата в примера заявка.*

След това да използвам някой инструмент за построяване и изпращане на единични заявки(като Postman), за да възпроизведа същата заявка или да я изменя така че да мога да я използвам за целите дори на атака от тип SQL инжекция, както е илюстрирано на фигура 2.3.1.5.:



*Фигура 2.3.1.5. Представяне на симулацията на примерната заявка чрез използване насофтуера Postman*

1. Използване на акаунт за базата данни с по-малко права

С цел ограничаване да даден атакуващ от извършване на злоупотреба, едно възможно решение е приложението да използва акаунт с по-малко привилегии така, че да не може да инжектира код, който при изпълнението си да навреди на базата данни – например да изтрие дадена таблица или цяла база. Това става просто като в конфигурационния файл на самото приложение зададем параметрите(потребителско име и парола) на клиент на базата данни без администраторски права и дефинираме действия със самата нея, които той има право да извърши. Този подход е изключително удобен за превенция на атаки тип SQL инжекция,чиято цел е изтриване на цялата база данни или конкретна таблица от нея, но не би бил толкова ефективен при предотвратяване на някой от другите представени видове. Причината за това е, че операции, характерни за другите видове атаки от този тип, като например извличане на всички редове от дадена таблица, не се считат за опасни и тази мярка би ги пропуснала като надеждни. По този начин атакуващият ще успее да достигне до търсената от него информация, без въобще да бъде „усетен“.

1. Използване на системни запазени процедури

Системните запазени процедури представляват предварително компилирани последователности от SQL твърдения, подередени в транзакции, с присвоено име, които могат да бъдат използвани от различни приложения. Използването им може да подпомогне контрола на достъпа до данните, при което се запазва тяхната цялост, и да подобри продуктивността на самата програма, защото твърденията в запазената процедура се съставят само веднъж. Други предимство на тези процедури е, че те се намират на едно място - в базата данни, тъй че при промяна на някоя от тях, всички клиенти на базата данни ще се обръщат към новата версия на този код. По този начин се спестява необходимостта от актуализация на устройствата на всички потребители. Нека да разгледаме пример за това какво представлява примерна запазена процедура. Нека имаме заявка от типа:

SELECT Name, HeadOfState FROM Country

WHERE Continent = con (23)

Същата тази заявка(23) може да бъде заменена от следната запазена процедура(24)

DELIMITER //

CREATE PROCEDURE country\_hos

(IN con CHAR(20))

BEGIN

SELECT Name, HeadOfState FROM Country

WHERE Continent = con;

END //

DELIMITER ; (24)

Тук в дефиниция(24) имаме запазената команда за създаване на процедура – CREATE PROCEDURE следвана от даденото и име – в случая country\_hos. След това следва списък от входните параметри на процедурата – тук само един con от тип символен. След това с BEGIN означаваме началото на тялото и, което в случая е заявката от по-горе, и завършва с END. Примерно извикване на процедура има следния вид(25):

db.query('call test.country\_hos;').on('row', function(r) {... (25)

По този начин имаме възможността да извикаме запазена процедура без специална функционалност за това.

В контекста на софтуерната сигурност и в частност атаките от тип SQL, тези процедури имат няколко предимства. Първото е че те служат за предварително дефиниране на кода, който ще се подаде на сървъра, а след това приемат параметри за тази заявка. По този начин, сървърът разграничава входящите данни и дори при евентуален опит за атака от типа на въвеждане на username = ‘’ OR 1=1’, то подаден на процедурата този параметър ще бъде интерпретиран изцяло като стрингова стойност и ще се търси потребител, чието име напълно съвпада с подадената стойност. Така може да се постигне сигурност, че никакъв злонамерен код няма да бъде изпълнен. Второто предимство се състои във възможността при изпълнението на дадена заявка да се върне обратно предишното състояние на базата в случай на грешка в изпълняващата се заявка. Така може да се контролира изпълнението и да се възстановят евентуално нанесени щети, независимо дали са неволно или не.

Проблемът на този прийом се състои в това, че цялата обработка на пристигащите данни се извършва на сървъра, на който се намира базата данни, а това крие своите рискове.

1. Предварително приготвени заявки

Този прийом е много подобен на използването на запазени процедури с тази разлика, че тук обработката на данните се извършва в приложението на сървъра, а не от интерпретаторът на базата данни. Както и при вече разгледания тип методология за защита от атаки от тип SQL инжекция, тук предварително дефинираме общия вид на заявката и след това подаваме данните, които са пристигнали от потребителският интерфейс. И тук стойностите в тази информация ще бъдат интерпретирани като текст и евентуален злонамерен код би бил анализиран просто като низ. Тук, за разлика от запазените процедури, обработваме информацията, пристигаща от потребителя на сървъра, на който се намира приложението.

1. Създаване на резервно копие на базата данни

Както разгледахме в третия случай на атака от тип SQL инжекция по горе, възможно е евентуална такава да доведе до изтриване на цялата база данни и по този начин загуба на ценна и поверителна информация. В такъв случай, единственото възможно решение на проблема е да се създаде резервно копие на информацията на сървъра, на който се намира тя. Повечето днешни имплементации на сървъри, на които се намират бази данни, поддържат такава функционалност и е въпрос на следване на предварително подготвени инструкции. Препоръчително е често да се правят резервни копия на базата данни с цел при евентуална атака да се възстанови максимално скорошна версията на съхраняващата се инфoрмация.[14]

Въпреки, че това би спомогнало връщането на изтрита информация във определен вид, то не е превантивно средство срещу извършване на евентуална атака. То не служи за спиране на злоупотреби, а по-скоро за смегчаване на последствията от успешна такава.

1. Съществуващи решения за справяне с атаки от тип SQL инжекция(практически реализации)

Освен подходите за превенция на атаки от тип SQL инжекция, които разгледахме в секция 2.3., съществуват и алгоритми за тази цел. Макар и те да са много и най-различни, не е възможно да обхванем всички. Затова ще се фокусираме върху два от тях поради ефективността им и сходността им с нашия алгоритъм. Те са алгоритъма за предотвратяване на атаки от тип SQL инжекция чрез използване на разпознаване на шаблони на Кхарче, Патил, Гохад, Амбеткар и Метод за откриване на атаки от тип SQL инжекция, базиран на премахване на атрибути на SQL заявката на Лий, Жеонг, Йео, Муун. Първият от двата използва основно алгоритъма на Ахо-Корасик за откриване на аномалии в даден текст. Затова и на него ще посветим отделна глава. Нека сега разгледаме трите алгоритъма в следващите секции ги разгледаме по-подробно в следващите секции.

1. Алгоритъм на Ахо-Корасик

Алгоритъмът е измислен от Алфред Ахо и Маргарет Корасик и се използва за разпознаване на различни шаблони от дадено множество в даден текст с произволна дължина. Неговата идея е по предварително изградено множество от низове и шаблони да намери техните срещания в целевия текст. Една от неговите най-важни характеристики е, че той активно използва краен детерминиран автомат с цел да разпознае даденият текст или шаблон от вече споменатият списък. Нека сега да разгледаме самият алгоритъм. Той се състои от две фази:

Фаза 1: Подготвителна фаза

По време на тази фаза се дефинират основните инструменти, които се използват в нашия алгоритъм, а именно автоматът(краен и детерминиран) и неговите функции при неуспех и на изхода. Затова условно можем да я разделим на три етапа:

Етап 1: Конструиране на краен детерминиран автомат

По време на тази фаза, ние създаваме гореспоменатият автомат, който ще използваме за разпознаване на предварително зададените шаблони след това.

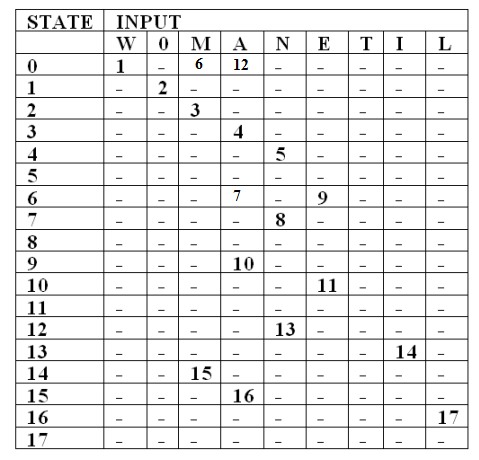
Състоянията на този автомат ще имат за имена последователни числа, преходите ще са надписани с буквите, които съставят шаблоните. Нека разгледаме един опростен пример:

Нека имаме множество от шаблони S={WOMAN, MAN, MEAT, ANIMAL}. Тогава крайният детерминиран автомат разпознаващ това множество има следния вид, показан на фигура 2.4.1.



*Фигура 2.4.1. Примерен автомат, използван от реализация на алгоритъма на Ахо-Корасик*

А неговата функция p на преходите има следния вид(фигура 2.4.2.):



*Фигура 2.4.2. Функция на преходите на гореспоменатия автомат*

Тоест за дадено състояние, например 6 имаме p(6, ‘A’)=7 и p(6, ‘E’)=9.

Така представен вече имаме общия вид на нашият автомат. Този алгоритъм обаче използва още една подробност на крайните детерминирани автомати и тя е функцията при неуспех f.

Етап 2: Конструиране на функцията при неуспех f

За нея е характерно, че тя се извиква когато при прочитане на следващият символ от големия низ няма възможен преход, тоест ако се намираме в състояние q и следващият символ от входия низ е ‘a’, то !p(q,’a’), тоест p не е дефинирана. В такъв случай извършваме преход към състояние f(q). Най-общо можем да дефинираме функицята при неуспех като най-дългата наставка на даден шаблон, която е представка на даден друг. Целта на това изображение да подсигури това, че няма да разгледаме даден символ повече от веднъж. Нека разгледаме горния автомат с функцията при неуспех, илюстриран на фигура 2.4.3.:



*Фигура 2.4.3. Функция при неуспех за примерния автомат*

А табличният вид на f показан на следващата фигура 2.4.4.:



*Фигура 2.4.4. Табличен вид на функцията при неуспех*

Крайните състояния на дадения краен детерминиран автомат са част от множеството {5, 8, 11, 17}.

Етап 3: Конструиране на функцията на изхода o

Функцията на изхода o e дефинирана върху множеството от крайните състояния на автомата (в конкретния случай {5, 8, 11, 17}) и ги изобразява в шаблони от даденото множество. В нашият пример табличният вид на o изглежда така(фигура 2.4.5.):



*Фигура 2.4.5. Функцията за разпознаване*

С това завършва подготвителната фаза на алгоритъма, по време на която дефинираме крайния детерминиран автомат.

Фаза 2: Търсене

Това е фазата, при която вече дефинираният алгоритъм се прилага върху конкретен текст или низ с цел откриване на някой от предварително дефинираните шаблони. Хубавото на този алгоритъм е, че благодарение на вече дефинираната функция при неуспех, този процес е линеен и представлява четене символ по символ на входния текст и извършването на съответните преходи в автомата на алгоритъма. Нека разгледаме един пример на анализ на следния стринг ‘WOMANETIMEAT’. Обработката му използвайки алгоритъма е показана на фигура 2.4.6.:



*Фигура 2.4.6. Обработка на примерен низ от алгоритъма на Ахо-Корасик*

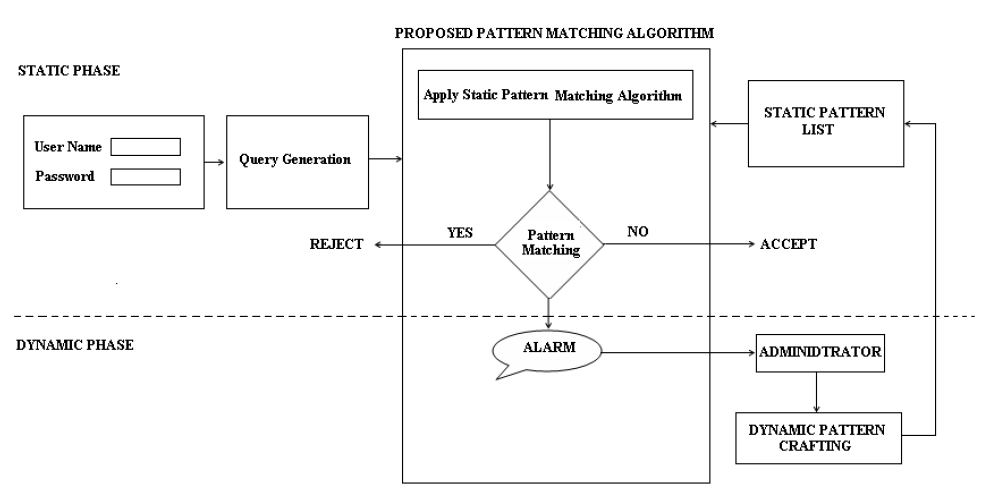
При всяко преминаване през крайно състояние, ние отчитаме чрез функцията на изхода шаблоните, чието срещане е установено. На края на изпълнението на тази фаза от алгоритъма, ние получаваме списък с всички шаблони, които сме срещнали при прилагането му.

Сред приложенията на този алгоритъм са за откриване на проникване, за установяване на плагиазъм, в биоинформатиката и за ефективен анализ на текстове. Първото представлява система, която събира информация от различни модули и източници в рамките на един компютър или мрежа и установява евентуални прониквания в нейната сигурност независимо дали те са от външен източник или е погрешно използване на вътрешено за организацията лице. Установяването на плагиазъм и ефективния анализ на текстове се отнасят към обработката на голямо количество текст и откриване на шаблони в него.

В нашият случай, този алгоритъм ще изиграе голяма роля при анализирането на низ съдържащ WHERE клаузата на заявката за някой от предварително зададените шаблони в нашия списък.

1. Алгоритъм за предотвратяване на атаки от тип SQL инжекция чрез използване на разпознаване на шаблони ()

Това решение е измислено от Кхарче, Патил, Гохад, Амбеткар [26]. Архитектурата му има следния вид:



*Фигура 2.4.2.1 Архитектура на имплементация на алгоритъма*

Предложената схема на фигура 2.4.2.1. се състои от два модула – статична фаза и динамична фаза. В списъка със статичните шаблони ние пазим всички шаблони на аномалии, които ще търсим в целевия стринг. Той е и основната част от статичната фаза. В нея ние ще проверяваме SQL заявката, която се изпраща от потребителя, посредством статичния алгоритъм за откриване на съвпадения. От своя страна в динамичната част при поява на нова аномалия, веднага се известява за откриването и и се създава нов шаблон за нея. След това той ще бъде прибавен към статичния списък с шаблони на аномалии. Нека сега по-подробно да разгледаме алгоритъма:

**Стъпка 1:** Генерира се заявка от тип SQL от данните получени от интерфейса на потребителя. Тя след това се изпраща за проверка от статичния алгоритъм за разпознаване на шаблони. Шаблон, от своя страна, наричаме последователност от определени символи, която представя общия вид на няколко текста. Той представлява текстовия вид на даден регулярен израз без допълнителните функционалности, които предоставя. Затова казваме, че нашия алгоритъм разпознава шаблони, а не конкретни низове, защото той разпознава общия вид. За да го обясним нагледно имаме следния пример: Нека искаме да разпознаем следните стрингове: ‘1 AND 2=3’, ‘1 AND 3=2’, ‘1 AND 4=5’ и така нататък. Вместо да проверяваме за наличието на всеки един от тях, ние ще търсим общия им вид или шаблон – ‘1 AND [0-9]+= [0-9]+’. Нека разгледаме неговия псевдокод:



*Фигура 2.4.2.2 Псевдокод на* процедурата по обхождане на списъка с аномалии

Този алгоритъм от фигура 2.4.2.2. е изпълнен под формата на два метода – един, който изпълнява процедурата по обхождане на списъка с аномалии, който сме показали горе, и друг, който е помощен за първия. Последния имплементира алгоритъма за разпознаване на шаблони , който разгледахме в секция 2.4.1.



*Фигура 2.4.2.3 Псевдокод на алгоритъма на Ахо-Корасик*

Аргументи на метода за разпознаване от фигура 2.4.2.3. са заявката и статичният списък от шаблони на аномалии с дължина m. След това за всеки един от шаблоните на аномалии извършваме следните операции:

1. Прилага се алгоритъма на Ахо-Корасик от Фигура 2.4.2.3.
2. След това се проверява дали резултатът от него е върнал индекс на срещане на съответния шаблон на аномалия.
   1. Ако е върнал индекс, то заявката се отхвърля.
   2. В противен случай се изчислява оценка за аномалии на съответната заявка.
      1. Ако тя е по-голяма от даденият праг, то се алармира до администратора на системата и заявката се изпраща за по-нататъчна обработка от него. След получаването на такава аларма, по-нататъчен анализ за наличие на аномалия се извършва ръчно. При откриване на такава, се съставя нов шаблон и се прибавя към статичния списък за проверка на съвпадения.
      2. В противен случай, заявката не съдържа опасен код и може да се продължи към изпълнението и.

Нека сега обърне внимание на изчисляването на оценка за аномалии на генерираната заявка.Както вече споменахме ако по време на проверата с даден шаблон от статичния списък се постигне 100% съвпадение, то сме сигурни, че дадената заявка е „заразена“ с SQL инжекция. В случай, че не бъде открито съвпадение, се изчислява оценка на аномалия, която представлява дължината на подниза, който да отговаря на следното условие – не е 100% съвпадение, но е най-дългото съвпадение на част от някой от шаблоние. След това тази стойност се сравнява с дадения праг, който е 50% в случая, и при открито съвпадение се алармира и се изпраща към администратора.

Нека сега разгледаме един пример за предотвратяване на атаки от тип SQL инжекция използвайки този алгоритъм.

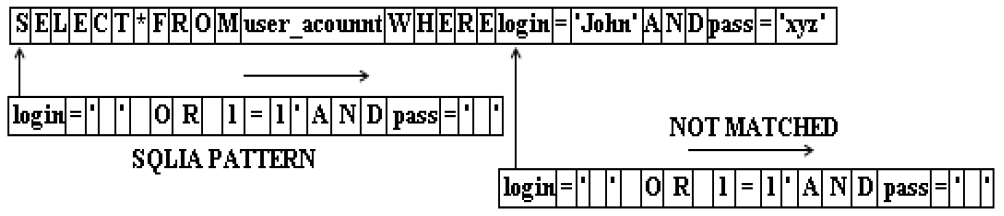
Нека една заявка, която не е опасна, да има следния вид:



Нека също така един от шаблоните от списъка да има следния вид:



В такъв случай прилагането на алгоритъма от по-горе върху конкретната заявка ще има следния вид:



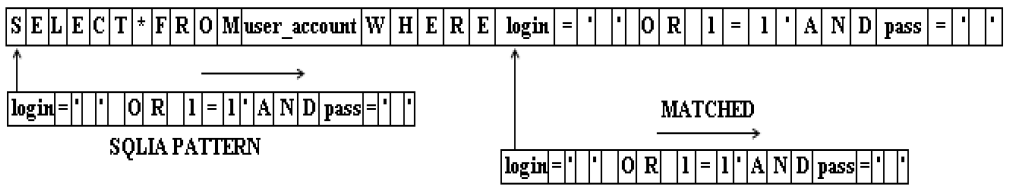
*Фигура 2.4.2.4. Визуално прилагане на алгоритъма върху примерна заявка*

Тук не се открива съвпадение и оценката на аномалия е ниска, затова заявката се одобрява и се изпълнява.

Нека сега да разгледаме заявка, която съдържа код, който отговаря на описанието на атака от тип SQL инжекция за извличане на информация, а именно:



Тук имаме твърдение от вида … OR 1=1…, което разгледахме по-горе. В този случай алгоритъма ще открие съвпадение с дадения шаблон, както е показано на фигура 2.4.2.5.



*Фигура 2.4.2.5 . Визуално прилагане на алгоритъма върху примерна заявка*

Така алгоритъма установява настъпване на атака от тип SQL инжекция и прекратява изпълнението и.

1. Метод за откриване на атаки от тип SQL инжекция, базиран на премахване на атрибути на SQL заявката

Измислен от Лий, Жеонг, Йео и Муун, основната цел на този алгоритъм е откриване на атаки от тип SQL инжекция чрез методите на статичния и динамичния анализ.[21] В основата си той представлява премахване на стойностите на дадени атрибути от текста на заявката по време на изпълнението на приложението(динамична част) и сравняване на обработената заявка с предварително приготвена такава(статична част). Нека сега да раазгледаме основните символи, които ще използваме в алгоритъма, записани в таблица 2.4.3.1.:

|  |  |
| --- | --- |
| Символ | Описание |
| I{t,f } |  |
| f | Функция, която връща заявката без стойностите на атрибутите. |
| FQ | Фиксирана заявка на приложението. |
| DQ{t,f } | Динамично генерирана заявка от потребителя. DQt е нормална заявка, а DQf съдържа код с атака от тип SQL инжекция |
| FDQ | Резултат от прилагането на функцията f върху FQ. |
| DDQ{t,f } | Резултат от прилагането на функцията f върху DQ{t,f}. |

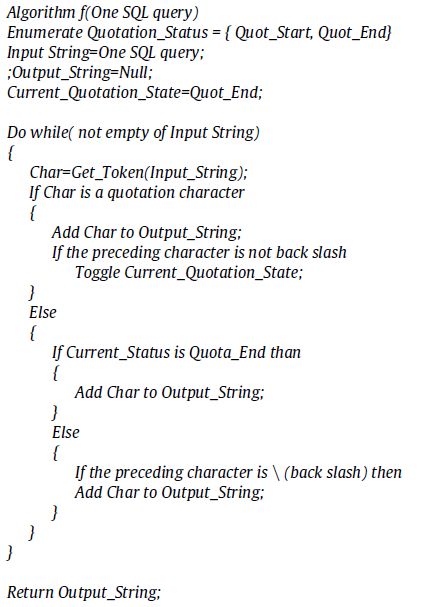
*Таблица 2.4.3.1. Основни термини в алгоритъма*

Методът за откриване използва функцията f за премахване на стийностите на дадени атрибути от заявката. Тя е показана като формула (26):

FDQ = f(FQ ), DDQ = f(DQ). (26)

Ще разгледаме две части на съответния алгоритъм, като ще представим подробно всяка една от тях:

В първата представяме функцията f, която премахва само текст от заявката, ограден с единични кавички(‘) след “=” или в скоби. В общия случай стойност на даден атрибут може да има следния вид: име\_на\_променлива = ‘текстова стойност’ или име\_на\_променлива = числова стойност. В случай на използване на функция в дадена заявка, извикването и е или във формата на име\_на\_функция(числова стойност) или име\_на\_функция(‘текстова стойност’). С цел да не се получи объркване, единичните кавички в даден стринг, заобиколен от кавички, които са част от текста за търсене се избягват с помощта на символа \ и не се вземат предвид при изпълнението на алгоритъма. Преди да преминем към представянето на методa като псевдокод, ще дефинираме функцията Get\_Token взима първия символ от входния стринг и след това го връща. Нека сега разгледаме и самия него на фигура 2.4.3.1.:

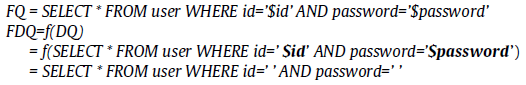


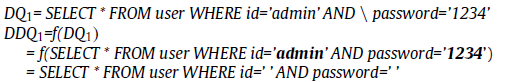
*Фигура* *2.4.3.1. Функцията f*

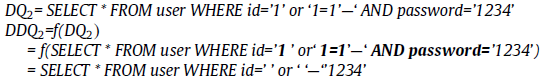
Нека сега разгледаме функцията по-подробно.

1. *Докато не сме дотигнали до празен входен стринг.*
   1. *Вземаме първия символ от него.*
   2. *Ако той е кавичка, то:*
      1. *Прибавяме го към резултата*
      2. *Ако предишния символ не е \, то сменяме стойността на текущото състояние на кавичките Current\_Quotation\_State.*
   3. *В противен случай:*
      1. *Ако текущата стойност на Current\_Quotation\_State е Quota\_End, то прибавяме символа към изходния такъв.*
      2. *В противен случай ако предишния символ е \, то прибавяме символа към крайния резултат.*
2. *Връщаме крайния стринг.*

Нека сега разгледаме прилагането на функцията f върху трите различни заявки от таблицата по-горе на фигура 2.4.2.2.:







*Фигура 2.4.3.2. Прилагане на дефинираните функции върху входен низ.*

Нека сега да сравним получените резултати от прилагането и да дефинираме формула (27):

|  |  |
| --- | --- |
|  | (27) |

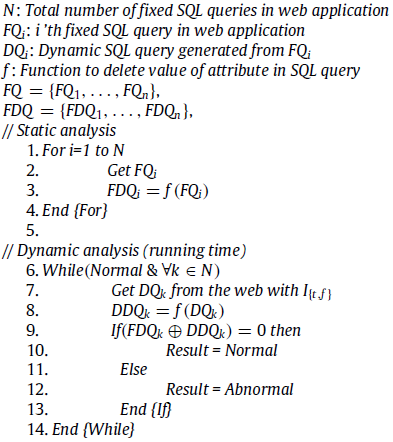
В случая с  се има предвид оператор, който връща 0 ако двата символа са еднакви и единица, ако са различни. По този начин, прилагайки оператора върху два низа като резултат получаваме число в двоичен вид, за което е в сила, че ако десетичния му вид е равен на 0, то низовете са еднакви, а в противен случай – са различни. Съответно, тъй като FDQ е очакваният вид на заявката, ако вида на DDQ след прилагане на f при прилагане на  ако върне 0, то заявката е нормална, а иначе – не. Нека разгледаме нашия пример, илюстриран 2.4.2.4.:



*Фигура 2.4.2.4. Прилагане на* 

И така установяваме, че DDQ1 е нормална, а DDQ2 съдържа опасен код.

Нека сега да разгледаме втората част на алгоритъма. За нея ще използваме вече разгледаната функция f. Нека разгледаме и самия метод на фигура 2.4.3.5.:



*Фигура 2.4.3.5. Представяне на обобщения вид на f.*

Тук вземаме предвид няколко параметъра:

* N – броя на различните заявки, които нашето ще изпрати към базата данни.
* FQ – списъка с всички заявки FQ1 ,FQ2 ,…,FQN, които приложението би генерирало към базата данни.

Нека сега опишем и този метод:

1. *За всяка заявка FQi намираме стойността и след прилагането на f.*
2. *Докато резултатът от изпълнението на текущата стъпка е Normal и имаме заявка от даденото множество, но с генерирана потребителска информация повтарямв следните стъпки:*
   1. *Намираме стойността DDQi и я сравняваме с FDQi.*
   2. *Ако съвпадат, то връщаме като резултат Normal.*
   3. *Иначе Abnormal.*
3. Сравнителен анализ на решенията

Нека сега да анализираме разгледаните алгоритми и да ги сравним. За първият алгоритъм от раздел 2.4.1. можем да кажем, че макар и да съдържа много добри идеи за инструменти, които да бъдат използвани, той има една слабост и тя е начинът, по който изглеждат шаблоните. Ако те са твърде подробни, както е използваният в горния пример



то има голяма вероятност да бъдат пропуснати важни случаи, като например:



Тук проверката за горния шаблон няма да върне открито съвпадение. Това има за цел да обърне внимание на евентуалните потребители, които биха искали да внедрят този алгоритъм да внимават, когато създават шаблоните си. Успешен вариант за шаблон тук би бил



Или дори по-добър вариант като например следния: [0-9]+=[0-9]+. Който би открил и съвпадения от вида на 2=3, 0=0 и т.н.

Именно вида на шаблоните е пряко свързан със структурата на разглежданият автомат. Ако шаблоните са твърде подробни като представяне, то автоматът би бил твърде сложен и непрактичен за поддръжка поради многото си състояния и преходи. Една такава машина, която би трябвало да съхранява шаблоните в посоченият вид



би трябвало да има поне 20 състояния, за да може да разпознае само дадения пример, а отделно поне 10 пъти по толкова състояния, за да считаме, че би открил най-основните нередности в дадена заявка. Един такъв автомат би бил огромен като структура и твърде объркан, за да може да бъде поддържан нормално. Това със сигурност ще се наложи, тъй като самото естество на алгоритъма е да бъдат откривани и нови шаблони на атаки от тип SQL инжекция. Този модул изисква участието на администратор на системата, който да проверява постоянно заявките с голям процент на вероятност да съдържат зловреден SQL код и да добавя нови шаблони за тях. В общия случай това би довело до проблеми, произтичащи от големия размер на използваната логическа структура.

Друг недостатък на този алгоритъм е фактът, че макар и да използваме автомат, чиято структура идваща от алгоритъм на Ахо-Корасик губи допълнително време, за да итерира през списъка от шаблони. Това има ключова за алгоритъма роля, но все пак довежда до загуба на време и до по-голяма времева сложност. Както ще установим по-нататък нашият алгоритъм се основава единствено на алгоритъма на Ахо-Корасик и неговата сложност е , където N е броя на състоянията на автомата, а M е максималната разклоненост на дадено състояние от последния. Тук се намесва и броят на шаблоните от списъка, който използваме и ако го означим с L, то сложността на първия алгоритъм би била , където X е дължината на подадената заявка. Причината за това е, че преди да използваме алгоритъма на Ахо-Корасик, ние първо намираме всички срещания на първия символ на всеки от шаблоните от предварително приготвения списък.

Вторият алгоритъм е много по-ясен и не включва някакви структури, които да използваме за анализ на входния стринг, а само алгоритъм за премахване на части от него. Това означава, че единствените итерации са по символите на входния текст на заявката. Така можем да заключим, че времевата сложност на алгоритъма е , където X е дължината на подадената заявка. В това отновение този алгоритъм е много по-бърз от предишния.

Основният проблем на този алгоритъм се състои в това, че при анализа на заявката не се взема предвид какъв тип атака се опитва да извърши потребителят, който я изпраща. Фокусът тук е върху предотвратяването на атака от тип SQL инжеция от който и да е вид. Това обаче би било проблем в случай, че дадена компания би искала да има различно отношение към потребители извършили различни видове атаки от гореспоменатия тип. За да могат да имплементират това, разработчиците на фирмата ще се наложи да усложнят алгоритъма с допълнителни проверки.

Основните критерии, по които ще сравняваме, ще са Времева Сложност, дали откриването на атаки в заявката е автоматично или не, дали са необходими модификации на кода, дали разпознава тавтологии(твърдения от гореспоменатия тип 1=1), дали разпознава неправилни заявки, заявки с обединение, заявки за извличане на метаинформация за сървъра на базата данни и дали разпознава извикване на функции от типа на if, benchmark и т.н. Нека сега разгледаме една таблица, в която сме изброили тези критерии и съответно резултатите от анализа:



Таблица 2.4.3.1. Сравнителен анализ на двата представени алгоритъма

Както виждаме при алгоритмите, използващи автомати, всичко зависи от това какъв автомат ще се използва и от това колко богат е езикът му. За сметка на това, за вторият алгоритъм е без значение това, защото той сравнява низове, а не използва специални инструменти за разпознаване. Също така модификации в кода на алгоритъма не са необходими при нито един от алгоритмите, но при алгоритмите, използващи автомати, е необходимо да се правят промени в неговата структура, така че да може да открива повече аномалии в заявката.Засичането на атаката не е автоматично само при първия алгоритъм, защото при него се изсква ролята на администратор на системата, който да проверява съмнителни заявки и да заключава дали те са реални атаки или не.

1. Изводи

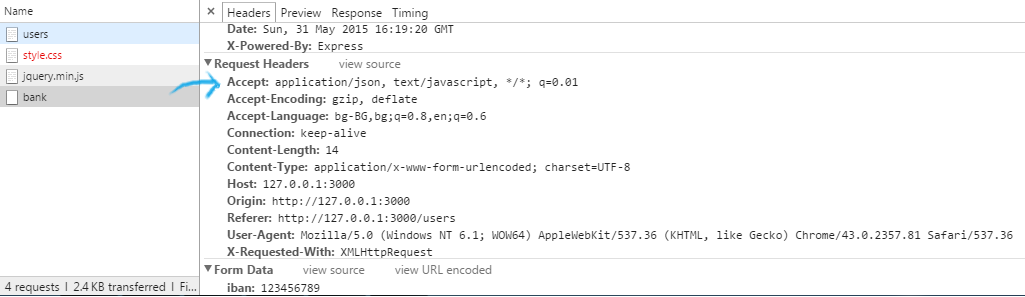
В заключение можем да кажем, че съществуват алгоритми за предотвратяване на атаки от тип SQL инжекция и те са подходящи за използване в зависимост от целта, която искаме да постигнем. Двата разгледани от нас са доста различни един от друг, но имат своите приложения. Като цяло алгоритъма, който ще предложим в следващите глави се базира основно на алгоритъма на Ахо-Корасик, който е в оснаовата и на първия от разглежданите алгоритми. Като предимство на това, което ние ще представим, е бързина и конкретна имплементация на автомата, предложен в описанието на горе споменатия. Вторият алгоритъм е по-универсален в предотвратяването на атаки от тип SQL инжекция. Неговият недостатък е, че не успява да разграничи видовете атаки и по-този начин ще предотврати дори атаки от тип сляпа SQL инжекция , но без да даде допълнителна информация за вида и, което е част от целта на дипломната работа. . Именно поради това. считаме, че алгоритъма, който ще предложим ще бъде достатъчно бърз и в същото време ще може да разграничи различните видове атаки от тип SQL инжекция.

1. Алгоритъм за откриване на аномалии

Нека сега представим нашия алгоритъм за откриване на аномалии, базиран на вече разгледания Алгоритъм на Ахо-Корасик. Основното в алгоритъма, който предлагаме, е предсказване на атака от тип SQL инжекция чрез откриване на аномалии в потребителското поведение, която съхраняваме в базата данни на нашата система. Затова ще се съсредоточим върху първия тип атаки от тип SQL инжекция – слепи инжекции, защото именно те са най-добрият индикатор, че даден потребител би извършил злонамерено действие срещу базата данни. Причината за това е, че когато даден недоброжелател иска да атакува дадена база данни без да има пряк достъп до нея, той трябва да установи типа и преди да извърши самото нападение. Разбира се, в някои случаи това е лесно за установяване, например, ако приложението е написано използвайки ASP.NET фреймуърка, то има голяма вероятност зад него да стои MSSQL база от данни, но това не винаги е така и не е в сила в общия случай. Нека за начало да обърнем внимание на основата на нашия алгоритъм, а именно алгоритъма на Ахо-Корасик, и след това да продължим към разглеждането на цялата логика.

1. Описание на алгоритъма

Както вече споменахме под аномалии в потребителското поведение ще имаме предвид атаки от тип сляпа SQL инжекция. Основния начин, по който може да бъде извършена такава атака е чрез изпращане на HTTP заявка, чиито данни съдържат SQL код, който при съставянето на SQL заявка, ще бъде изпълнен и ще измени очакваният резултат като ще съдържа информация за базата данни или за таблицата в нея. Нека да разгледаме отново един пример за това как би станало това. Нека нормална заявка в нашето приложение има вида:

 *Фигура 3.1.1.Общ вид на разглежданата от нас заявка*

Тук на фигура 3.1.1. тялото на заявката съдържа едно своиство, а именно iban. След като тя достигне до сървъра, ще стигнем до създаването на следната заявка(28) към базата данни:

SELECT BD.withrawal, I.IBAN, B.bankName

FROM diplomna\_rabota.bank\_data BD

LEFT JOIN diplomna\_rabota.ibans I ON I.Id = BD.IBAN

LEFT JOIN diplomna\_rabota.banks BON B.Id = I.bankId

WHERE I.IBAN = 123456789 (28)

Ако даден потребител реши да извърши атака от гореспоменатия тип спрямо тази заявка, то той ще промени стойността на полето iban в HTTP заявката като в примера, показан на фигура 3.1.2.:

*Фигура 3.1.2. Заявката с променена стойност на полето iban*

Сега той ще очаква, че получената SQL заявка ще има вида(29):

SELECT BD.withrawal, I.IBAN, B.bankName

FROM diplomna\_rabota.bank\_data BD

LEFT JOIN diplomna\_rabota.ibans I ON I.Id = BD.IBAN

LEFT JOIN diplomna\_rabota.banks BON B.Id = I.bankId

WHERE I.IBAN = 123456789 OR 1=1 (29)

И като резултат ще върне всички редове от тези таблици. По този начин атакуващият ще разбере, че системата е уязвима на атаки от тип SQL инжекция.

Нашият алгоритъм цели да открие именно подобен тип аномалии в потребителските заявки.

Затова при съставянето на съответната SQL заявка, ние изпълняваме алгоритъма на Ахо-Корасик върху нея. Така попаднем ли във финално състояние на автомата на последния, то имаме случай на аномалия или сляпа SQL инжекция. Затова алармираме и не се стига до изпълнение на заявката. Нека да разгледаме описанието на нашия алгоритъм:

1. *Съставяме SQL заявката към базата данни от данните, пристигнали от интерфейса.*
2. *Прилагаме алгоритъма на Ахо-Корасик върху новосъздадената заявка. Задаваме на временна промелива returnedResult стойност -1. Тя ще съхранява резултатът, който алгоритъма ни ще върне и започваме самото му изпълнение:*
   1. *Ако при изпълнението на алгоритъма достигнем до финално състояние на автомата:*
      1. *Задаваме като стойност на returnedResult индексът н символа, който ни е довел до крайното състояние, и връщаме returnedResult.*
      2. *Прекратяваме изпълнението на функцията за изпращане на заявката към системата за управление на бази данни и тя няма да се изпълни.*
      3. *Връщаме алармиращ текст, че е отчетена аномалия.*
   2. *В противен случай:*
      1. *Стойността на returnedResult няма да се промени и при достигане на края на текста на заявката ще върнем -1.*
      2. *Продължаваме изпълнението на функцията за изпращане на заявката към системата за управление на бази данни и тя ще се изпълни. Връщаме резултатите от базата данни.*
3. Модификация на алгоритъма на Ахо-Корасик

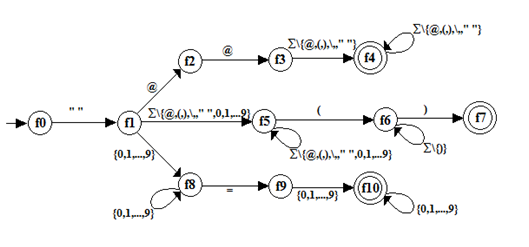
Както вече споменахме в една от по-горните глави, основната роля в алгоритъма на Ахо-Корасик играе автоматът, който използваме. Именно той определя езикъ или множеството от шаблоните, които ще разпознаваме в даден текст. Ние направихме анализ на аномалиите в потребителското поведение и се спряхме на следните основни аномалии в дадена потребителска заявка:

* Тавтологии или грешни SQL заявки – това са заявки или от типа “… OR 1=1”, или – “… AND 2=3”. Поради това решихме да ги обединим в първия подвид шаблони за аномалии в потребителското поведение, а именно .
* Научаване на информация за базата данни чрез извличане на стойността на свойства на сървъра, като @@version – за версията, @@datadir – за главната директория на базата данни и @@hostname – името на сървъра, на който се намира базата данни. Те заедно оформят втория шаблон – “ @@[adehimnorstv]”. Тук нарочно сме избягали от конкретизиране на определените ключови думи, защото за нас е важно да хванем шаблона, който започва със @@.
* Последната група аномалии е извикване на вградени функции и процедури, като например IF(), BENCHMARK(), DATABASE(), VERSION(). Чрез тях достигаме и до третия шаблон – “ [abdehimnofrstv](...)”. Тук, по подобие на предишния случай, за нас е по-важно наличието на “(…)”, отколкото думата преди него, затова конкретизираме до буквите, съставящи горните думи.

На база на горния анализ достигаме до следната азбука на целевия автомат:



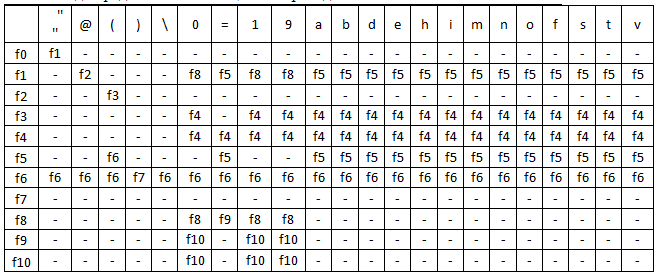
Нека сега да разгледаме и самият автомат:

*Фигура 3.2.1. Общ вид на автомата, който използваме*

На горната фигура 3.2.1. сме представили общия вид на нашият автомат, който ще използваме, за да установим наличие на анималии в данните, пристигащи от интерфейса.

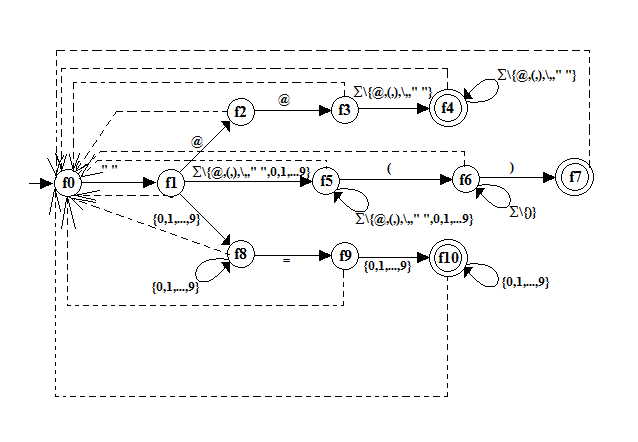
Както виждате, началното състояние на този автомат сме означили с f0. Тъй като всички шаблони от гореспоменатите типове започват с интервал, то логично е първият преход да разчете именно този символ от азбуката. При срещането му, автоматът преминава към състояние f1. Именно то е мястото, от което трите случая на нашия алгоритъм се разделят. Най-горния преход от f1 до f2 разпознава символа @ и оттук започва да разпознава шаблоните от тип “ @@[adehimnorstv]”. От своя страна, преходът от f1 до f5 ще бъде извършен само при срещане на символ от азбуката, различен от @, (, ), \, числата от 0 до 9 и интервал. Така това разклонение ще разпознава шаблона “ [abdehimnofrstv](...)”. Накрая, но не на последно място, преходът от f1 до f8 разпознава числата от 0 до 9. Така ще открием срещане на първия шаблон - .

Нека сега да представим и таблицата на преходите на този автомат, показана на фигура 3.2.2.



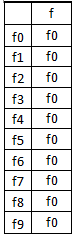
Фигура 3.2.2. Табличен вид на функцията на преходите

Тук обаче важно е да споменем, че както в алгоритъма на Ахо-Корасик, така и при нас, ще използваме функцията при неуспех, за да можем да реализираме обхождането на подадения низ без да се налагат допълнителни излишни проверки. При нас тя ще изглежда така обозначена на фигурата 3.2.3. на автомата:



*Фигура 3.2.3. Общ вид на функцията при неуспех*

Очевидно функцията при неуспех в нашия случай ни връща отново в състояние f0, тоест табличният и вид изглежда по този начин(фигура 3.2.4.):



*Фигура 3.2.4. Табличен вид на функцията при неуспех*

3.3. Сравнение със съществуващите решения

Вече разгледахме два алгоритъма в секции 2.4.1. на Кхарче, Патил, Гохад, Амбеткар(за по-кратко Алгоритъм 1) и и секция 2.4.2. на Лий, Жеонг, Йео и Муун(Алгоритъм 2) и сега е време да ги сравним с представеният от нас.

Нека първо да пресметнем времевата сложност на нашия алгоритъм. Както вече говорихме, сложността на алгоритъма на Ахо-Корасик и неговата сложност е , където N е броя на състоянията на автомата, а M е максималната разклоненост на дадено негово състояние. Като вземем предвид и дължината X на подадената заявка, получаваме като краен резултат . Нека сега разгледаме таблицата 3.3.1. за сравнение на трите алгоритъма:



Таблица 3.3.1. Сравнение на трите разглеждани алгоритъма

Нека сега сравним нашия алгоритъм с двата вече съществуващи:

Както вече споменахме нашия алгоритъм използва алгоритъма на Ахо-Корасик за разпознаване на шаблони, подобно на Алгоритъм 1. Основната слабост на последния е начинът, по който изглеждат шаблоните. При нас този проблем е решен благодарение на използването на по-общ вид на всеки от разпознаваните стрингове от нашия алгоритъм. Също така предложеният от нас алгоритъм има и по-добра времева сложност.

Ако трябва да го сравним с Алгоритъм 2, нашият алгоритъм е по-бавен във времево отношение, но го превъзхожда с това, че успява да разграничи атаките от тип сляпа SQL инжекция от останалите типове. Това е и основната причина, поради която смятаме, че алгоритъма ни е по-добър от Алгоритъм 2.

1. Използвани технологии

Инструментът, който ще създадем с цел илюстриране на предложения в секция 3 алгоритъм, представлява уеб приложение, тъй като те са най-застрашени от евентуални атаки от този тип. Основните езици в приложението са JavaScript и HTML, като HTML използваме за статичната част на интерфейса на нашето приложение, а JavaScript за динамичната и сървърните части. За сървърната част използваме пакета Express.js[28] над платформата Node.js[27], приложенията над която се пишат чрез езика JavaScript. За статичната част на браузърния код на приложението сме използвали генератор на шаблони за HTML код, наречен Jade. Базата данни, с която приложението комуникира е MySQL и комуникацията с нея се осъществява посредством модули. Нека сега обърнем по-специялно внимание на всяка от тези технологии.

1. Сървърна част

Node.js е публично достъпна система с отворен код, която се поддържа от различни платформи. Приложенията, които създаваме чрез нея са написани на Javascript, но благодарение на Node.js платформата за изпълнение могат да бъдат пуснати на операционни системи като OS X, Microsoft Windows, Linux, FreeBSD, NonStop и IBM i. Тази система използва виртуалната JavaScript машина Google V8, за да изпълни написаният код. В допълнение на това, Node.js предоставя включена библиотека, която да симулира уеб сървър без да е необходимо използването на софтуери като Apache HTTP Server или IIS.

Едно от основните предимства на Node.js като сървърна платформа е възможността да се използват вече имплементирано модули, които са написани или от самият разработчик или от други потребители на платформата. По този начин не се налага да имплементираме самостоятелно дадена функционалност и след това да установяваме дали има грешки в нея и как да ги поправим. Тук на помощ идва специално създадената система за управление на пакетите, наречена npm, която помага на програмиста лесно и без особени усилия да прибави вече написаният и тестван модул към своето приложение. Също така кодът за всеки един от тези пакети се съхранява в хранилището за код github, където има функционалност за обявяване на проблеми и грешки в поведението и имплементацията му. Така потребителите, ползвали даден модул, могат да дадат своето мнение или оплакване от реализацията му и то да достигне веднага до неговия разработчик, който има време да оправи съответните проблеми и да качи нова версия на своя продукт. Още едно положително нещо на npm мениджъра на пакети е възможността да се постави рейтинг на всеки от тях. Така, когато някой търси модул, има възможността веднага да види и настоящата му оценка и лесно да прецени дали той ще му свърши работа или не . Повечето пакети съдържат в описанието си и кратко обяснение как да бъдет добавени и използвани след това от потребителя, който иска да го внедри в приложението си. Добавянето на npm мениджъра е част от инсталирането на Node.js системата.

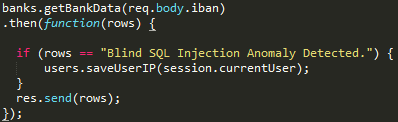
Един от възможните модули, които могат да бъдат добавени и използвани е Express. Той предоставя вече написани функционалности, които улесняват още повече използването на логиката на Node.js за създаване на HTTP сървър.

Напоследък технологията добива все по-голяма популярност като платформа за сървърната част и се използва от големи корпорации като Microsoft, Yahoo, SAP, PayPal и други.

4.1.1. Отложено изпълнение на функции

Една от най-важните характеристики на node.js като сървърен език е неговата асинхронност. Това означава, че не се изчаква изпълнението на дадена функция, за да се пристъпи към следващата, а директно след извикването на първата преминаваме към втората.

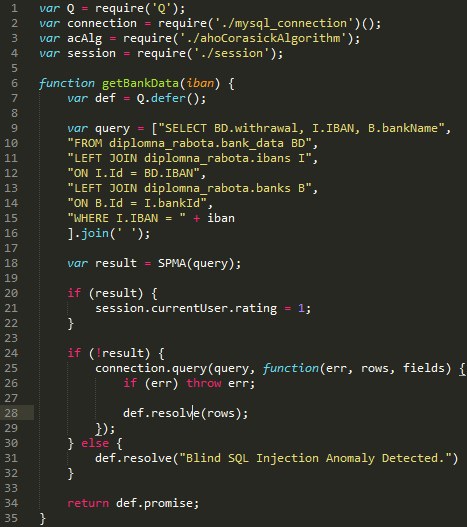
В повечето случаи, обаче, ние бихме предпочели да изпълним последователно инструкциите в нашия код. Именно заради това на помощ ни идват вече предифинираните от други потребители модули. Нека вземем най-популярния сред тях, а именно q. Благодарение на него ние можем да покажем последователност от функции, които да се изпълнят една след друга, като например в следната извадка от нашия код, демонстрирана на фигура 4.1.1.1.:



*Фигура 4.1.1.1. Пример за отложено изпълнение на функции*

Тук, както виждаме на фигура 4.1.1.1., Извикването на функцията getBankData от модула bank е последвано от извикването на метода .then, която приема като аргумент следващата функция от последователността, а именно тази за обработването на резултата. Така ще преминем към изпълнението на втората функция чак когато свърши това на първата - bank.getBankData.

За да можем да обясним поведението на .then нека разгледаме тялото на функцията bank.getBankData, което представяме на фигура 4.1.1.2.:

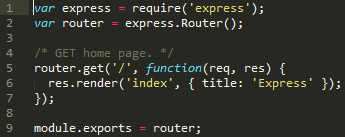


*Фигура 4.1.1.2. Тялото на функцията getBankData*

Показали сме и модулите, които ще използваме в нашия модул за извличане на банкова информация, за да се види, че сме прибавили модула q. Причината да можем да изпълним функцията .then от фигура 4.1.1.2. е че като резултат от изпълнението на първата функция не получаваме дефакто резултата от нея, който е списък с банкова информация, а обект, който е наречен promise. Каква е процедурата? В началото на съответния метод ние дефинираме отложения обект( в нашия случай def), който получава стойността Q.defer(). Оттук нататък до края на изпълнението на функцията нейното поведение ще се диктува и от този обект. Например нека разгледаме случая, в който изпулнението на метода е достигнало до ред 25 и следва изпълнението на заявката от връзката с базата данни(за справка – секция 5.3.). Тук веднага щом се извика функцията connection.query ще се пристъпи към изпълнението на ред 30 и т.н. ще стигнем до края на функцията. В обикновения случай най-вероятно вече ще сме излезли от нея и ще сме прекратили отдавна изпълнението и преди да са върнати резултатите от SQL заявката. В такъв случай ще се загуби идеята на нашата логика. Затова вместо да върнем какъвто и да е резултат ние връщаме въпросния обект promise. При него идеята е чрез използване на безкраен цикъл да се забави изпълнението на следващата функция чак докато не върнем резултат. Последното става щом приключи изпълнението на извикана в рамките на текущи метод функция def.resolve или def.reject. В нашия случай това са редове 26, 28, 31. При първата се подава обект, който служи като резултата на функцията, както в нашия код е резултата от SQL заявката или съобщението за открита аномалия. Целта на втората е да се обяви, че по някаква причина е настъпила грешка в изпълнението.След като се изпълни една от двете операции, ще се прекрати въпросния безкраен цикъл и ще се продължи към следваща функция.

4.1.2. Рутери

Рутерите са основният начин за определяне на това как да отговори приложението през крайните точки на сървъра на заявка от клиента. Комуникация между сървъра и клиентите се извършва посредством HTTP заявки към определен URL. Рутера е комбинация от URI, HTTP метод и една или повече функции, които да биват извиквани в тази крайна точка. Нека сега да разгледаме дефиницията на дадена крайна точка на сървъра. Тя има следния вид: router.METHOD(path, [callback...], callback). Тук router е модула express, METHOD е метода на HTTP заявката, path е путя на сървъра и callback е функцията, която ще се изпълни при съвпадение на path с URL на заявката. Нека сега да покажем пример за това от кода на нашето приложение:

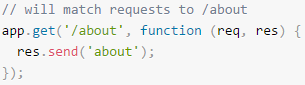


*Фигура 4.1.2.1.Общ вид на рутер от нашето приложение*

На фигура 4.1.2.1. виждаме дефиниция на дефиниция на крайна точка за сървъра. Първо започваме с дефинирането на инстанция на модула express, именно express. Оттук вече вземаме рутер обект от нея – router. След това следва гореописаната конструкция. Тук тя има следното тълкуване – ако потребителят изпрати HTTP заявка с метод GET до url ‘/’, то ще се изпълни функцията за изпращане на index.jade страницата. Тук е мястото да споменем, че поддържаните от Express HTTP методи са: get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search, и connect. Също така има още един специален метод all, чието извикване в горната конструкция ще изпълни съответната функция независимо от типа на HTTP заявката.

Следващият елемент в дефинирането на крайна точка, който ще разгледаме е пътя. В дефиницията на последната той може да бъде просто текст, шаблон на текст или регулярен израз. Нека сега разгледаме примери за всеки един от тях.

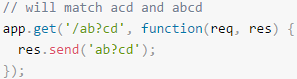
Първият тип – като обикновен текст - е най-рапространеният. Пример за него е:



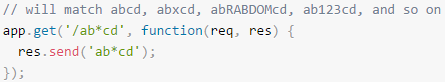
Фигура 4.1.2.2. Дефиниция на крайна точка

На фигура 4.1.2.2. виждаме крайна точка, която разпознава единствено път /about.

Вторият разглеждан от нас вид път е като шаблон на текст. Нака за целта разгледаме няколко примера:



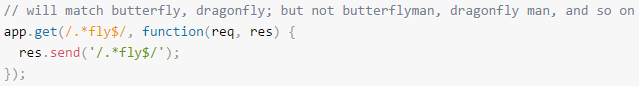
Фигура 4.1.2.3. Дефиниция на крайна точка



Фигура 4.1.2.4. Дефиниция на крайна точка

На фигура 4.1.2.3. виждаме пример за шаблон от вида а?, където а е даден символ. Тук чрез него ще се разпознават два текста като път, а именно abcd и acd. На следващата фигура 4.1.2.4. вече виждаме шаблон \* и значението му в случая е разпознаване на каквато и да е последователност от каквито и да са символи стига да започва с ab и да завършва с cd.

Последния тип от тези, които ще споменем, е регулярни изрази. За целта нека видим следния пример:



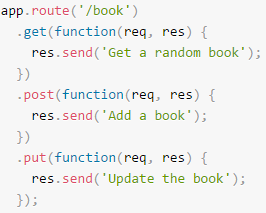
Фигура 4.1.2.5. Дефиниция на крайна точка

Тук(фигура 4.1.2.5.) виждаме следния регулярен израз - .\*fly$/. Неговата цел е да разпознае всички текстови оследователности, които имат поне 4 символа и завършват на fly. Както е указано в коментара над кода на фигура 3.1.2.5. това могат да са думи като butterfly, dragonfly и други.

За да завършим разглеждането на дефиницията на дадена крайна точка ще споменем, че при изпращане от страна на клиента на заявка до определен път можем да имаме и повече от една функции които да се изпълнят. Тук обаче е важно да споменем, че ако по време на изпълнението на някоя от тях настъпи грешка, това ще доведе до прекратяване на изпълнението и следващите функции в последователността няма да се изпълнят.

Нека сега да обърнем малко внимание на така наречените рутер, които първо показахме на фигура 4.1.2.1. Когато искаме да отделим определени функционалности в общ модул или като отделна логика, ние искаме да ги разграничаваме по някакъв начин. Тук на помощ идват два приома, които ни помагат да групираме определени крайни точки чрез пътищата им.

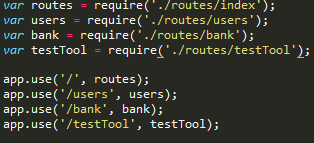
Първият от тях е функцията route(), която е представена на следния пример:



Фигура 4.1.2.6. Дефиниция на множество от крайни точки

Така показаната конструкция на фигура 4.1.2.6. ни показва дефиницията на група от крайни точни на свързване, които имат общ път – book, но са със съответно GET, POST и PUT като HTTP методи.

Другият начин е въпросният, показан на фигура 3.1.2.1., а именно класът express.Router, който се използва за създаване на подрутер на основния под формата на модул. Поради тази причина той има вида на стандартен рутер на приложение със своите дефиниции на крайни точки на свързване. Дефиницията на такъв рутер е показана на фигура 4.1.2.1., а използването му ще покажем на следващата фигура:



Фигура 4.1.2.7.Пример за дефиниция на крайни точки от нашия код

Тук ако например имаме крайна точка в модула banks и тя има път /testTool, то тя ще може да бъде достъпена чрез заявка с URL адрес /banks/testTool.[24]

1. Клиентска част

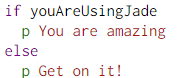
За реализацията на клиентската част ще използваме езика Jade който дава възможност за бързо и удобно писане на HTML шаблони. За него са характерни най-вече три неща: написаният код ще бъде транслиран като HTML, поддържа се динамичен код и предизползваемост. С първото се има предвид, че кодът, който сме написали в даден файл, при изпълнение ще бъде превърнат в чист HTML. Предимството тук обаче е минималистичния подход при писане на код. За разлика от стандартните HTML файлове, тук не се налага да пишем отварящи и затварящи тагове, да се грижим дали сме поставили последния. Също така основните атрибути на HTML елементите не е необходимо да бъдат описани под формата на име=“стойност“, а има други механизми. Ще илюстрираме казаното дотук със следния пример:



*Фигура 4.2.1. Пример за дефиниция на jade документ*

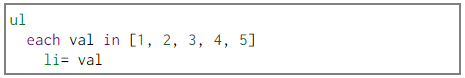
Тук на фигура 4.2.1., както виждаме, вместо използването на отварящ и затварящ таг на всеки елемент, обозначаването на принадлежността се извършва чрез нови редове и отстъпления. Също така типа на всеки таг се изписва само веднъж като обикновен текст. Самият интерпретатор на кода ще разбере, че трябва да сложи затварящ таг щом срещне друг елемент със същото или по-малко отстъпление от началото на реда. Както вече говорихме, задаването на клас и идентификатор на даден елемент не е необходимо да става както досега, а именно <div id="container" class="col">. Тук единственото, което трябва да напишем, е #container.col, а интерпретаторът сам ще разбере, че това са тези атрибути и ще ги зададе на елемента по подразбиране, а именно div.

В този пример можем да видим и още една от характеристиките на езика Jade, а именно неговата динамичност. Един от основните начини за зареждане на дадена Jade страница е като я върнем като резултат от дадена заявка и я заредим като страница. Той също така ни дава и правото да изпратим и обект, който съдържа така нареченият контекст на съответната страница. Благодарение на него, при интерпретирането на кода и превръщането му в чист HTML можем да извършваме проверки, да итерираме по списък от обекти, който също е част от контекста и други. Нека илюстрираме това с горния пример. В него можем да видим следната конструкция:



*Фигура 4.2.2. Откъс от дефиниция на jade документ*

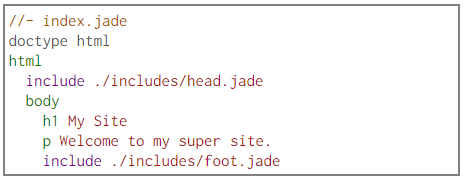
Тук(фигура 4.2.2.) ние извършваме проверка дали контекста съдържа свойство youAreUsingJade. Ако да, то връщаме едно HTML съдържание, а ако не – друго. Нека да видим и пример за итериране на списък от обекти:



*Фигура 4.2.3. Откъс от дефиниция на jade документ*

В примера, илюстриран на фигура 4.2.3., създаваме динамично неподреден списък, като обхождаме елементите на подадения масив(в случая съдържащ числята от 1 до 5 включително) и за всеки един от тях създаваме li елемент, чието съдържание е стойността на съответното число.

Последната характерна черта на този език е преизползването на код. То се поддуржа и в стандартните HTML файлове, но тук е опростено като писане на код. Да си представим, че пишем дадена страница, използвайки Jade, но искаме да вмъкнем да определено място съдържанието на друг файл. Тогава единственото, което трябва да направим е да напишем следното:



*Фигура 4.2.4. Откъс от дефиниция на jade документ*

Тук на фигура 4.2.4. имаме файл с име index.jade, в който прибавяме преди тялото съдържанието на файла includes/head.jade, а в края му – това на includes/foot.jade. Така не се налага да пишем един и същи код много пъти, а и самото му прибавяне е изключително ясно и подредено.

В заключение можем да кажем, че основното предимство на Jade е простия и ясен код, който трябва да представя една страница. Благодарение на това е по-малко вероятно да се объркаме дали сме спазили отварящите и затварящите тагове на нашите HTML елементи, защото интерпретатора се грижи за това.

Тази е и едната причина, поради която използваме Jade като средство за създаване на интерфейсната част от нашия туул. Другата е, че модула Express.js предлага опростен начин на работа с него и лесно внедряване в логиката на сървърната част.[22]

1. Проектиране
2. Модел на данните

Нека да разгледаме диаграмата на базата данни, показана на фигура 5.1.1., и да опишем всяка една от нейните таблици:

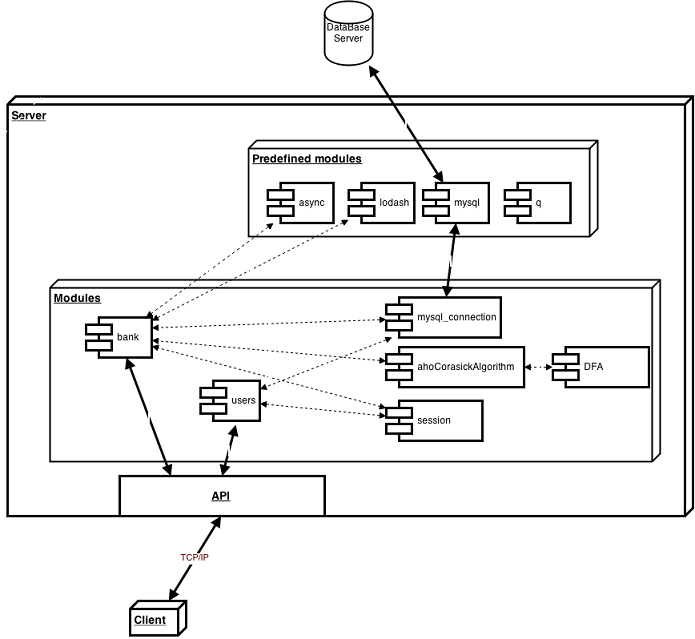
 *Фигура 5.1.1.*

Както виждаме тя се състои от пет таблици, като отляво стоят таблиците, които пазят информация, свързана с потребителите на системата, а отдясно – тези, касаещи симулираните данни, които ще се извличат от приложението и които ще са нашият обект на атака.

Нека да започнем с вторите. За целите на нашето приложение ще симулираме информационна система за извличанията от различни банкови акаунти на даден потребител. За тази цел създаваме три таблици. Първата от тях съхранява имената на банковите институции(banks), втората – IBAN номерата на различните сметки в системата, а третата – връзката между даден потребител, неговите IBAN номера и сумите, които са теглени от тях.

Таблиците отляво са същността на нашият инструмент. Първата от тях – user\_data, съхранява потребителското име и паролата на всеки един от тях. Втората, а именно client\_status, има централна роля в нашия алгоритъм. Тя свързва даден потребител с IP адресите на устройства, от които той е влизал в системата и рейтинга на всяка една то тези комбинации(потребител и IP адрес). Рейтингът може да има две стойности – 0 или 1. Ако за дадена комбинация той е 1, то съответния клиент на системата влизащ през устройство с дадения адрес е ненадежден и е показал някоя от анализираните по-горе аномалии в поведението си, а в противен случай – комбинацията е наред. При вписването на даден потребител, освен проверката на легитимността на въведените от него потребителско име и парола, се проверява и IP адреса на устройството, от който той се опитва да влезе в системата. Ако вече го имаме запазено, то на сървъра се запазва информация за текущия потребител, а именно неговият IP адрес и рейтинг. Ако го нямаме като запис от таблицата client\_status, го прибавяме заедно с потребителското име и рейтинг, който е 0. Съответно по-нататък ще вземаме предвид неговия рейтинг при анализиране на заявките му.

1. Aрхитектура на системата



Фигура 5.3.1. Диаграма на архитектурата на нашето приложение

На фигура 5.3.1. сме представили архитектурата на нашето приложение. Както виждате основни елементи в него са клиента, сървъра и базата данни. Основно внимание в тази секция ще обърнем на сървъра и на структурата му. Както вече споменахме в секция 3.1.2., Express предоставя възможност за дефиниране на крайни точки за свързване и по този начин да бъде дефиниран приложно-програмен интерфейс или API, което да служи за осъществяване на връзката с клиента. От друга страна всяка една точка на свързване има метод от модулите bank и users който извиква при пристигнала заявка. Модулите на нашето приложение използват вече предефинирани модули. Най-важните от последните са async, lodash, mysql, q. За последния вече говорихме в секция 3.1.1. Модулът mysql използваме за комуникация с базата данни. В нашето приложение, както ще опишем в следващите параграфи, сме създали собствен модул, който управлява комуникацията с базата – mysql\_connection, който използва, обаче, вече предефинираният такъв.Следващият async ни трябва за пускане на няколко асинхронни процеса например в тестовия модул. Последният, lodash, е за обработване на колекции от данни в нашите модули.

Нека сега да разгледаме поотделно всеки един от нашите модули: users, bank, mysql\_connection, session, ahoCorasickAlgorithm.

Модулът users отговаря за операциите, свързани с вземане на информацията за даден потребител като рейтинг при подадени потребителско име и парола и запазване на статуса на даден потребител. Методите, които той предоставя за работа, са getUserData и saveUserIP, като те имат следните интерфейси, представени в следващите блокове код (27) и (28):

function getUserData(username, password) (27)

function saveUserInfo(user) (28)

Първият(27) получава потребителско име и парола и взема всички редове от таблиците за потребители, съдържащи тези стойности за съответните колони. Втората функция(28) запазва информацията за даден потребител, като за целта и се подава обект user, съдържащ новите стойности на съответните колони.

Модулът bank отговаря за операциите, свързани с вземане на информацията за тегленията от дадена сметка на потребителят, вписан в системата. Методите, които той предоставя за работа, са getBankData и getBankDataAsync, като те имат следните интерфейси, представени в следващите блокове код (29) и (30):

function getBankData(iban) (29)

function getBankDataAsync(iban, callback) (30)

Първата функция(29) получава номер на сметката(IBAN) и по него взема всички тегления от нея, като комбинира информацията от няколко таблици. Вторият метод(30) има същата функция като първия(29), но с тази разлика, че ще бъде използван за тестовия инструмент и по този повод му се подава и един допълнителен параметър, който да се извика при получаването на информацията, за да може да продължи изпълнението на следващата стъпка.

Модулът mysql\_connection отговаря за създаване на връзката към базата данни. Както вече споменахме, ще използваме набор от пренаписани модули за някои основни операции, като комуникация с базата данни и изпълняване на заявки, за отложеното извикване на функции и др. В случая използваме вече написаният модул mysql. За да се осъществят гореспоменатите операции, обаче, трябва да се създаде връзка към съответната база данни. Хубавото в случая е, че не е необходимо да се създава връзка за всяка заявка, а само веднъж по време на изпълнението на приложението. Затова сме отделили създаването на тази връзка в отделен модул, наречен от нас mysql\_connection, който просто трябва да прибавим към всеки модул, който ще комуникира с базата данни. Единственото, което този модул предоставя е самата връзка към базата.

Следващият модул, който ще разгледаме, е session. Той е най-малкия като размер и единствената цел, която има, е да пази информация за текущо вписаният потребител и да я предоставя на всички останали модули.

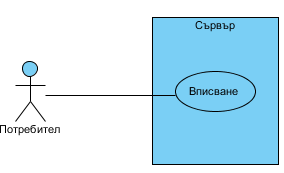
Най-важният модул в нашия инструмент е последния, който ще разгледаме – ahoCorasickAlgorithm. Той съдържа алгоритъмът за откриване на аномалии в потребителската заявка. За целта той съдържа два метода – един, който реализира алгоритъма на Ахо-Корасик и един за нашия алгоритъм. Тук е важно да споменем, че този модул използва един друг модул, който ние сме написали – DFA. Той съдържа реализацията на автомата, който описахме в секция 4.4., но не му отделяме специално внимание. Методът, който този модул предоставя има следния вид:

function SPMA(query) (31)

Този метод (31) получава като аргумент заявката, която ще анализираме и връща като резултат или индекса на срещане на аномалията, или -1 в случай, че не открием такава.

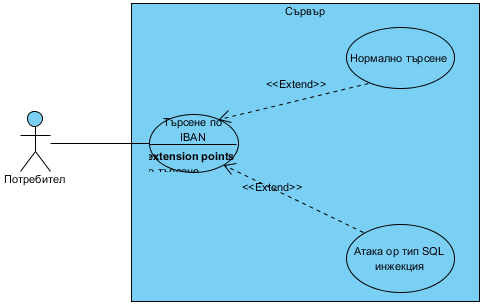
1. Потребителски сценарии

За тестовия инструмент можем да разграничим следните 3 потребителски сценария, представени на фигури 5.4.1., 5.4.2., 5.4.3. и на съответните им таблици със спецификации:



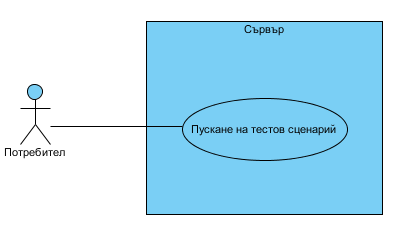
Фигура 5.4.1. Диаграма на потребителски сценарий 001: Вписване на потребител

|  |  |  |
| --- | --- | --- |
| **Идентификатор:** | 001 | |
| **Име:** | Вписване в системата | |
| **Актьори:** | | Потребител |
| **Описание:** | | Вписване на потребителя в системата посредством потребителско име и парола. |
| **Начало:** | | Натискане на бутона Login след като потребителят е въвел потребителско име и парола. |
| **Предварителни условия:** | | **Потребителят да не е вписан в системата.** |
| **Условия след приключване:** | | Потребителят да е вписан в системата и неговата информация за IP и статус да се пази в модула session. |
| **Нормален поток на действията:** | | 1. Потребителят въвежда данните във формата за вписване си и натиска бутона Login. 2. Ако случайно е оставил едно или и двете полета празни, системата ще го подкани да въведе стойност. 3. Данните, записани в двете полета се изпращат към сървъра посредством заявка с url /users. 4. Сървърът проверява съществува ли такъв запис в таблицата diplomna\_rabota.user\_data. 5. Ако да, зарежда данните за потребителя в сървърния модул session. 6. В противен случай изпраща отговор, че не съществува такъв потребител. |
| **Изключения:** | | Няма |
| **Специални изисквания:** | | Няма |



Фигура 5.4.2. Диаграма на потребителски сценарий 002: Търсене на банкова информация по IBAN

|  |  |  |
| --- | --- | --- |
| **Идентификатор:** | 002 | |
| **Име:** | Изпращане на заявка за извлеченията от сметка по IBAN | |
| **Актьори:** | | Потребител |
| **Описание:** | | Извличане на тегленията от дадена сметка на съответния потребител на системата |
| **Начало:** | | Натискане на бутона SEARCH във формата за търсене на съответната информация за сметка на потребителя. |
| **Предварителни условия:** | | **Потребителят да е вписан в системата.** |
| **Условия след приключване:** | | Няма |
| **Нормален поток на действията:** | | 1. Потребителят натиска гореспоменатия бутон SEARCH. 2. Ако не е въвел стойност в полето IBAN бива приканван да го направи. 3. В противен случай се изпраща въвдената стойност до сървъра чрез заявка към url /bank.   В сървърната част:   1. Съставя се заявката към базата данни. 2. Полученият текст на заявката се изпраща за обработка от алгоритъма ни за проверка за аномалии. 3. Ако бъде открита такава аномалия, не се стига до изпълнение на заявката. Към интерфейса се изпраща отговор, че е засечена аномалия в заяката и в модула session се задава стойност 1 на рейтинга на потребителя. 4. Ако не засечем такава аномалия, изпращаме заявката за изпълнение към базата данни. Резултатът от нея се изпраща към интерфейса и се визуализира в таблицата за резултата, |
| **Алтернативни потоци:** | | Няма |
| **Изключения:** | | Няма |
| **Специални изисквания:** | | Няма |



Фигура 5.4.3. Диаграма на потребителски сценарий 003: Пускане на тестов сценарий в системата

|  |  |  |
| --- | --- | --- |
| **Идентификатор:** | 003 | |
| **Име:** | Пускане на тестов сценарий | |
| **Актьори:** | | Потребител |
| **Описание:** | | Стартиране на тестов сценарий, чиято цел е да демонстрира действието на създадения алгоритъм. |
| **Начало:** | | Натискане на бутона Run Test. |
| **Предварителни условия:** | | 1. **Потребителят да е вписан в системата.** 2. **Да е отворен тестовия модел, тоест потребителят да е натиснал бутона TEST TOOL от главния интерфейс.** |
| **Условия след приключване:** | | Няма. |
| **Нормален поток на действията:** | | 1. Потребителят натиска бутона Run Test. 2. Изпраща се GET заявка с utl /testTool..   В сървърната част:   1. Генерират се 10 потребителя и IP-та, както и 10 заявки, по една за всека една двойка потребител и IP. 2. Всяка заявка от полученият списък се обработва от нашия алгоритъм за откриване на аномалии. 3. По познатия вече сценарий, ако открием аномалия, то запазваме като резултат съобщение за открита аномалия и променя стойността на рейтинга на съответния потребител в таблицата diplomna\_rabota.client\_status. 4. Като резултат изпращаме към клиента списък от обекти съдържащи мотребителското име на потребителя, IP адреса, заявката, която се преща за изпълнение, статуса на съответния потребител и резултатът от заявката. |
| **Изключения:** | | Няма |
| **Специални изисквания:** | | Няма |

1. Тестване.
   1. Реализация на тестовия инструмент

След като разгледахме алгоритъма откриване на аномалии и архитектурата на приложението, нека сега да обърнем внимание на тестовия модул. Най-общо за него можем да кажем, че целта му е проследяване на трафика до нашия сървър и откриване на аномалии в заявките. За целта сме реализирали отделен модул, наречен Security Traffic Analyzer. В него потребителят ще има възможността да стартира следенето на заявки и те ще се визуализират в таблица отдолу, заедно с техния резултат, информация и статус на потребителя, който ги изпраща.

За да можем да симулираме трафик към нашия сървър сме създали и отделно приложение за генериране на заявки, което също ще разгледаме, макар и по-повърхностно.

Накрая ще обърнем внимание и на модула за управление на конфигурациите за шаблони на дадени аномалии

Нека сега започнем с разглеждането на алгоритъма на тестовия туул.

* + 1. Алгоритъм

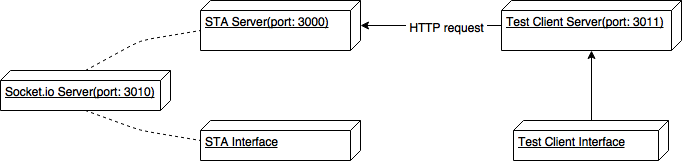
Алгоритъмът на тестовия модул използва нашия вече реализиран такъв за откриване на аномалии в потребителските заявки. Това, обаче, което е специфично и представлява най-голям интерес в него е начина на създаване на тестовите данни. Нека да разгледаме алгоритъма, а после ще го и анализираме:

1. *Потребителят отваря интерфейса за тестване чрез натискане на бутона TEST TOOL.*
2. *Потребителя изпраща заявка за генериране на тест(за справка виж потребителски случай 003).*
3. *Когато заявката стигне до сървъра, той започва да следи другите заявки, които постъпват към него и за всяка заявка пристигнала с url=‘/bank/’и HTTP тип post,:*
   1. *Ние изпълняваме вече имплементираната логика за проверка за аномалии от тип SQL инжекция и, ако не срещнем такива, я изпълняваме и връщаме резултата.*
   2. *Извличаме от тялото на заявката стойността на полето iban, а от заглавната и част IP адреса, потребителското име и статуса на съответния акаунт, който я е изпратил. Тези данни, заедно със резултата, изпращаме асинхронно към интерфейса.*
4. *Веднъж пристигнал до интерфейсната част на нашето приложение, полученият обект се визуализира в резултатната таблица.*

Основната цел на този алгоритъм е да покаже взаимодействието на различни потребители със системата, така че да може да се демонстрира работата и в нормални условия, макар и в случая да става въпрос за тестови данни.

* + 1. Архитектура

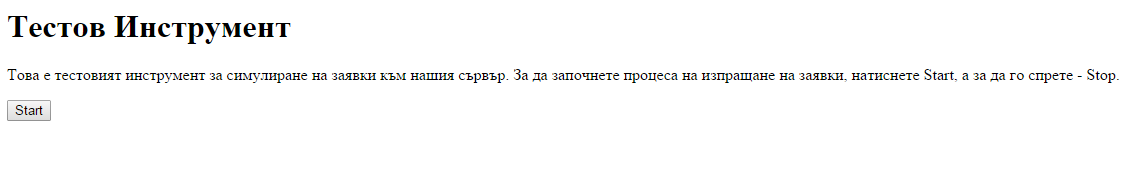
Нека да разгледаме общата архитектура на тестовия сценарии, за да можем да опишем след това всяка една стъпка от него:



Фигура 6.2.2.1.: Обща архитектура на нашия тестов сценарий

Както виждате на фигура 6.2.2.1., за нашия сценарии са ни необходими две приложения – едно, създаващо нашият сървър от реализацията, която описваме в глави 4 и 5, и друго, което да симулира работата на много клиенти с различни потребителски имена и IP адреси, които изпращат заявки към него.

Приложението, което симулира заявки, се нарича Test Client. То създава сървър, който слуша на порт 3011 и неговият интерфейс се състои от една страница, която има следния вид:



Фигура 6.2.2.2.: Главната страница на приложениетo Test Client

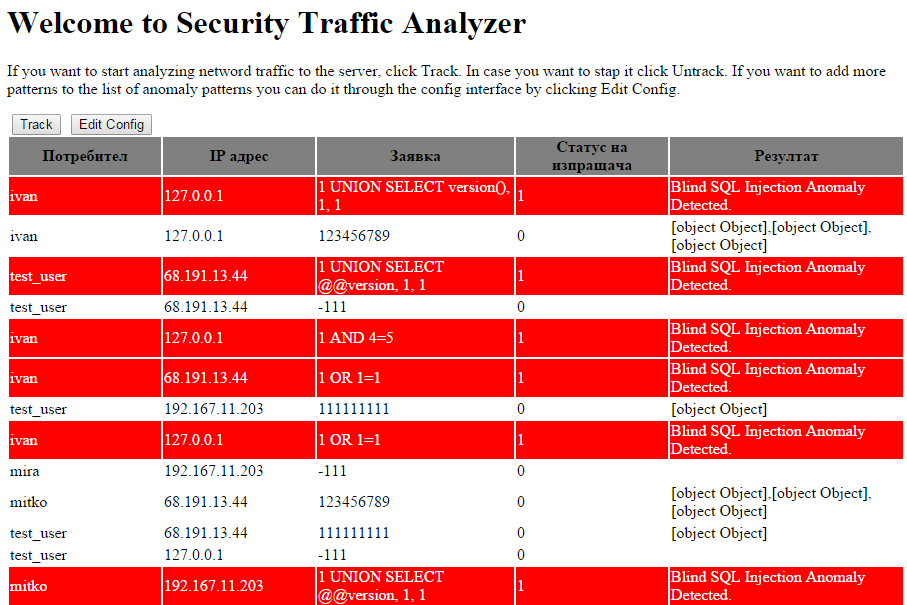
Както е видно на фигурата 6.2.2.2., страницата има един бутон, който при натискане изпраща заявка за стартиране на безкрайната процедура по генериране на HTTP заявки към нашия сървър, който ще наричаме Software Traffic Analyzer Server. След това текста на бутона ще се смени на Stop и при натискане от потребителя ще прекрати стартирания процес по изпращане на заявки и стойността на бутона ще стане отново Start. Нека сега разгледаме по-подробно логиката, която се изпълнява на сървъра.

При натискане на бутона Start потребителският интерфейс изпраща HTTL заявка към сървъра от тип GET с url ‘/req’. Когато тя пристигне на сървъра, той стартира безкраен цикъл с помощта на вече имплементираният модул infinite-loop. Неговата основна цел е имплементацията на логиката за безкраен цикъл, който може да се пуска и спира, както и да му се задава времеви интервал, през който да изпълнява зададеното му от потребителя тяло под формата на функция. В нашия случай създаваме инстанция на този модул, чиито времеви интервал е 3 секунди, а тялото му представлява създаване и изпращане на заявка към Software Traffic Analyzer Server.

Следващият елемент в архитетурата на тестовия сценарий е Software Traffic Analyzer Server или по-конкретно неговия модул Software Traffic Analyzer(STA). Алгоритъмът, който сме имплементирали в този модул е описан в предишната секция 6.2.1. За да може да изпрати асинхронно информацията за изпълнената заявка, ние ще използваме помощта на Socket.io сървър.

Именно Socket.io сървърър е и следващият обект на фигурата 6.2.2.1., който ще разгледаме. Той представлява вече разработена логика за асинхронна комуникация между два или повече клиента чрез използване на така наречените джобове или socket-и. В случая този сървър създава джоб на порт 3010 и слуша за два вида събития. Първия е подаване на информация, която се съхранява временно и след определен период от време се изтрива. Втория е свързаване с цел извличане на информация, която се съхранява. В случая двата клиента на Socket.io сървъра са Software Traffic Analyzer Server и интерфейсната част STA Interface. Логиката за асинхронност се осъществява чрез периодично изпращане на заявки за извличане на информация от интерфейсната част, която се съхранява в джоба на порт 3010.

Нека сега да обърнем внимание и на интерфейса STA Interface. Той се състои от две страници, главната от които има следния вид:



Фигура 6.2.2.3. Главната страница на STA Interface.

Както виждаме на фигура 6.2.2.3. тя се състои от два бутона – Track и Edit Config и таблица, в която ще визуализираме трафика към сървъра. При натискане на Track бутона ние стартираме изпращане чрез Socket.io на информация за постъпващи заявки към интерфейса, която след това се визуализира в таблицата. След това стартиране стойността на бутона става Untrack и при натискане спираме започнатия процес. Другия бутон Edit Config се използва за модифициране на тестовите данни, които използваме в нашия алгоритъм и по-подробно ще разгледаме тази процедура в Приложение 2: Добавяне на нови тестови данни.

1. Заключение

В началото на дипломната работа въведохме читателите в областта на софтуерната сигурност и по-конкретно на различните атаки срещу нея. След това разгледахме различни архитектурни подходи и два имплементирани алгоритми за предотвратяване на атаки от тип SQL инжекция. След анализиране и сравнение на последните два, установихме техните положителни и отрицателни страни. Адаптирахме един от разгледаните два алгоритъма и създадохме трети на базата на недостатъците на предишните, който да разпознава аномалии в потребителското поведение от тип SQL инжекция с достатъчно добра времева сложност. Също така, за да илюстрираме действието на нашия алгоритъм, проектирахме, реализирахме и валидирахме инструмент, работещ на локален сървър, онагледяващ действието му. Заедно с инструмента създадохме и имплементирахме тестов сценарии, който да показва действието на инструмента в близки до реалните условия. За целта въпросния сценарии включва друго приложение, което да симулира различни HTTP заявки от различни потребители и IP адреси. Като заключение бихме казали, че успяхме да изпълним задачите поставени в началото на дипломната работа.

Като предстояща работа по темата бихме посочили анализ и на други типове аномалии в потребителското поведение, като например изпращане на твърде много заявки към нашия сървър за кратко време( до 1 секунда). Това например би било индикация за намерението на съответния потребител да извърши нередности със системата и в частност атака от тип SQL инжекция. Има още много примери за евентуални аномалии в потребителската дейност, но те са в рамките на предположенията и засега няма реално разглеждане за тях. Именно това ще бъде налично в следващата версия на дипломната работа. Засега обаче представеният документ изпълнява поставените задачи на дипломната работа от секция 1.2..

Приложения

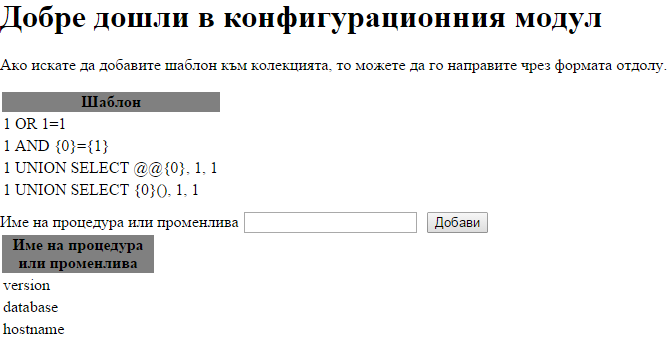
В тази глава ще представим двете приложения към нашата дипломна работа, а именно Приложение 1за инсталиране на нашия примерен инструмент и Приложение 2, съдържащо добавяне на нова дума към тестовите данни на тестовия инструмент.

Приложение 1: Инсталиране на примерния инструмент

Приложение 2: Добавяне на шаблон или ключова дума към тестовите данни

За да добавим нови думи към тестовите данни на нашето приложение, трябва да отворим конфигурациоония модул на предоставеният от нас тестов инструмент(за справки относно инсталирането виж Приложение 1). Най-лесният начиин за отварянето на този модул е като отворим страница с адрес <http://127.0.0.1:3000/config>. Другият вариант е чрез отваряне на главната страница на нашето приложение <http://127.0.0.1:3000>, където да се впишете като даден потребител, например username: mitko\ password: dimitar и оттам да натиснете бутона STA Tool и след това бутона Edit Config.

Щом отворим тази страница, ще видим 2 таблици – една за шаблоните и една за запазените имена на процедури или променливи. Информацията на двете таблици се взема от други две такива, намиращи се на сървъра, както е показано на фигура P.2.1:



Фигура P.2.1. Главната страница на конфигурационния модул.

Както е видно на Фигура P.2.1, имаме права само да добавяме имена на запазени процедури или променливи, но не и шаблони. Причината за последното е необходимостта от модифициране на автомата на Алгоритъма на Ахо-Корасик. Това може да се изпълнява само от администратор на приложението чрез добавяне на съответния запис в таблицата с шаблоните и променяне на преходите и евентуално прибавяне на състояния в автомата.

За да прибави име на променлива или процедура, е необходимо да я въведем в посоченото поле и да натиснем Добави. Новото име на запазена дума ще се появи в таблицата.

Използвана литература

[1] <http://phpsecurity.readthedocs.org/en/latest/Input-Validation.html>

[2] <http://www.w3schools.com/js/js_validation.asp>

[3] <http://verifyjs.com/>

[4] <http://www.the-art-of-web.com/html/html5-form-validation/>

[5] <http://www.intellectualtakeout.org/faq/2-how-are-cyber-attacks-carried-out>

[6] http://searchsoa.techtarget.com/definition/Web-site

[7] http://www.computerhope.com/jargon/w/webpage.htm

[8] http://searchsoftwarequality.techtarget.com/definition/Web-application-Web-app

[9]<http://www.acunetix.com/websitesecurity/web-applications/>

[10]<http://hackmageddon.com/category/security/cyber-attacks-statistics/>

[11]<https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf>

[12] <https://www.owasp.org/index.php/SQL_Injection>

[13] <https://www.owasp.org/index.php/Blind_SQL_Injection>

[14] <http://webcheatsheet.com/sql/mysql_backup_restore.php>

[15] <https://www.owasp.org/index.php/Top_10_2013-Top_10>

[16] <http://www.forbes.com/sites/jaymcgregor/2014/07/28/the-top-5-most-brutal-cyber-attacks-of-2014-so-far/>

[17] <https://blog.sucuri.net/2014/11/most-common-attacks-affecting-todays-websites.html>

[18] <http://whatis.techtarget.com/definition/Confidentiality-integrity-and-availability-CIA>

[19] <http://searchoracle.techtarget.com/definition/stored-procedure>

[20] Importance of Aho-Corasick String Matching Algorithm in Real World Applications, Saima Hasib, Mahak Motwani, Amit Saxena, International Journal of Computer Science and Information Technologies, Vol. 4 (3) , 2013, 467-469

[21] A novel method for SQL injection attack detection based on removing SQL query attribute values, Inyong Lee, Soonki Jeong, Sangsoo Yeo, Jongsub Moon,

Mathematical and Computer Modelling 55 (2012)

[22] JADE Language Documentation, http://jade-lang.com/

[23] Документацията на модула random-js

[24] <http://expressjs.com/guide/routing.html>

[25] A brief history of internet security, SC Magazine, September 24, 2009, <http://www.scmagazine.com/a-brief-history-of-internet-security/article/149611/>

[26] Preventing SQL Injection Attack Using Pattern Matching Algorithym, Kharche, Patil, Gohad, Ambetkar, Department of Computer Engineering, MAHARASHTRA INSTITUTE OF TECHNOLGY, 2013

[27] Node.js documentation. <https://nodejs.org/api/documentation.html>

[28] Express.js documentation. http://expressjs.com/4x/api.html