Съдържание

[Въведение в теорията на регулярните изрази 2](#_Toc483665765)

[Общ модел на атаката 4](#_Toc483665766)

[Атаки в .Net 6](#_Toc483665767)

[MatchTimeout параметър за превенция на „катастрофален backtracking” 15](#_Toc483665768)

[ReDoS атаки върху .Net MVC 17](#_Toc483665769)

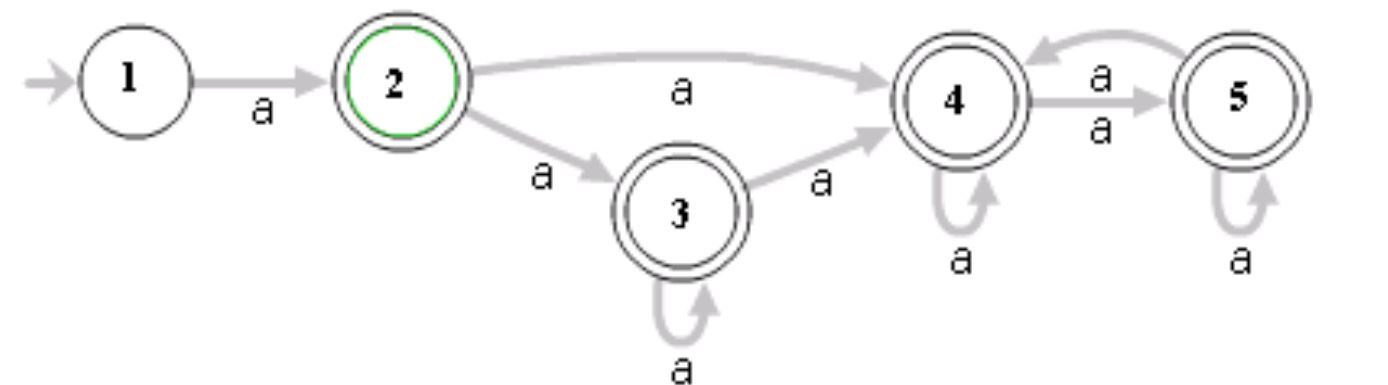
[Заключение за предпазване от атаки 24](#_Toc483665770)

[Източници 25](#_Toc483665771)

# Въведение в теорията на регулярните изрази

В настоящия доклад се разглежда темата за атаки от тип „отказ на услуга“ срещу регулярни изрази, които ще бъдат съкращавани понякога с английския термин ReDoS атаки, съкратено от Regular expression Denial of Service. Този тип атака се базира върху възможността да се използва факта, че имплементациите на регулярните изрази могат да попаднат в критични ситуации, при които започват да работят много бавно като под бавно се разбира експоненциално спрямо входа. Злонамереният потребител може да предизвика ситуация, в която подавайки точно определен вход срещу регулярния израз да накара системата да „увисне“.

За да стане ясно какъв е проблемът с регулярните изрази трябва да се разбере какъв е алгоритъмът, който решава задачата. Алгоритъмът е добре позната тема в математика и се осъществява с помощта на така наречените автомати (показан е пример малко по-надолу). Той строи недетерминистичен краен автомат, където за всяка двойка от състояние и символ има няколко възможни следващи състояния. Тогава алгоритъмът продължава като прави преминаване в друго състояние и това се повтаря докато не се достигне края на входа. Той изпробва всички възможни пътища ако се налага докато не намери съвпадение или докато не обиколи всички пътища и установи, че няма съвпадение. Например регулярният израз ***^(a+)+$*** се представвя чрез крайния автомат:



За входа ***aaaaX*** има 16 възможнипътя, както може да се проследи в автомата. За входа ***aaaaaaaaaaaaaaaaX*** от друга страна има 65536 възможни пътя и числото се удвоява за всяко допълнително **а.** Случаи като този са проблемни тъй като трябва да бъдат изминати много пътища преди да се разбере, че няма съвпадение. Това важи особено за наивни алгоритми решаващи задачата. Съществуват и ненаивни алгоритми, които могат да бъдат имплементирани добре и решението да е ефективно. Въпреки това проблемът се разпростира още повече като се има пред вид, че съвременните алгоритми се опитват да работят не само върху „чисти“ регулярни изрази, но и техни разширения, където се покриват специални условия.

Под „зъл“ регекс се разбира такъв, който може да спре да работи при конкретно подготвен вход. Примери за това са най-общо модели съдържащи :

* Групиране с повторение
* В повторената група:
  + Повторение
  + Смяна с припокриване

Конкретни примери за „зъл“ регекс са:

* (a+)+
* ([a-zA-Z]+)\*
* (a|aa)+
* (a|a?)+

Във всички тези примери забелязваме уязвимост към вход от вида: ***aaaaaaaaaaaaaaaaaaaaaaaa!***

# Общ модел на атаката

Атакуващият може да използва всички зняния за регулярните изрази и сравнително лесно да намери различни начини да навреди. Един вариянт е да потърси приложения, които използват регулярни изрази, съдържащи „зли“ регекси и да накара системата да „увисне“. Друг вариянт е ако самият регекс може да се повлияе от потребителския вход, атакуващият може да инжектира „зъл“ регекс и да направи системата уязвима.

Рискът за подобни атаки в уеб система е значителен, тъй като има много подходящи „елементи“ от системата, на които да се приложи, а и фактът, че при уеб приложенията често се налага потребителите да се регистрират, което изисква използване на регулярни изрази при обработката на входа, е много показателен за уязвимостта, която съществува в такива среди. Във всеки слой от уеб-а съществува голяма вероятност да се използват регулярни изрази, които съдържат „зъл регекс“. Атакуващият може да накара да забие не само уеб браузъра на персонален компютър или мобилно устройство, но и Firewall-a, както и дори уязвим Уеб сървър. Например ако програмистът използва регекс за валидация на клиента и той съдържа „зъл“ регекс, атакуващият може да предположи, че същият уязвим регекс се използва и на сървъра и да изпрати добре подготвен вход, с който да прекрати нормалната работа на сървъра.

Стандартен начин за извършване на ReDoS атака протича по следния начин:

* Отваряне на JavaScript
* Откриване на „зъл“ регекс
* Подготване на подходящ вход за атаката
* Изпращане на валиден вход през прокси
* Промяна на заявката, така че да съдържа злонамерен вход

Пример за това е код, който проверява при регистрация дали паролата, която въвежда потребителят се съдържа в потребителското име и ако е така не прави регистрация от съображения за сигурност. В тази ситуация ако атакуващият въведе изразът *^(([a-z])+.)+[A-Z]([a-z])+$* като потребителско име и *aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!* като парола системата ще забие.

# Атаки в .Net

От практическа гледна точка съществуват няколко причини за това едно приложение да е уязвимо към ReDoS атака. От една страна разработката на мобилни приложения и използването на АПИ-та учеличава броя на недоверените източници, от които идва информацията, а от друга липсата на достатъчно добро валидиране на входа. В основата на проблема лежи фактът, че голям брой програмисти не разбират напълно как работят регулярните изрази „зад сцената“ и използвайки неподходящ регулярен израз поставят цялото приложение на риск.

В следващите няколко параграфа се разглежда въпроса за това как се открива проблема още в ранен етап на разработката на едно .Net приложение и как се предотвратява използвайки стратегия на писане на Unit тестове.

Цялостната работа на модул за работа с регулярни изрази няма да бъде разгледана, но основните правила за откриване на текст, който е описан с регулярен израз са следните:

* Най-ранният (т.е този който се среща най-вляво) печели: регулярният израз се прилага към входа започвайки от първият символ и продължава към края на текста.
* Кванторите са лакоми: кванторите указват по колко пъти може да се повтаря нещо. Стандартните квантори се опитват да повторят всичко по възможно най-много пъти. Процесът на отказване от намирането на всички възможни повторения се нарича backtracking.

В предстоящите параграфи се използват недетерминирани крайни автомати за работа с регулярни изрази. Техния начин на работа се състои в следното: сравнява се всеки елемент от регекса с входния текст като се пази позицията, където пътя се е разклонявал и е бил направен избор между две състояния. Ако една опция се окаже неуспешна, алгоритъмът се връща към най-близката запазена позиция. Този алгоритъм е имплементиран в .Net, както и в много други съвремени езици. В .Net се използва namespace-a *System.Text.RegularExpression* и съответно класовете в него, които са имплементация на този алгоритъм и предоставят различни функционалности за работа с регулярни изрази.

Описаният вече алгоритъм има както положителна страна, така и негативна. Негативната се състои в това, че използването на backtracking-а работи много бързо, когато търси положително съвпадение, но когато става дума за отрицателно (входният текст не се разпознава от регулярния израз), този алгоритъм започва да работи много бавно. Това е така, тъй като алгоритъмът трябва да потвърди, че не съществува път, през който може да се мине, така че да се разпознае входа, т.е всички възможни пътища трябва да бъдат тествани. С обикновения подход, който прави максимален брой повторения откриването на отрицателни съвпадения става значително по-бързо, но положителните се разпознават много по-бавно.

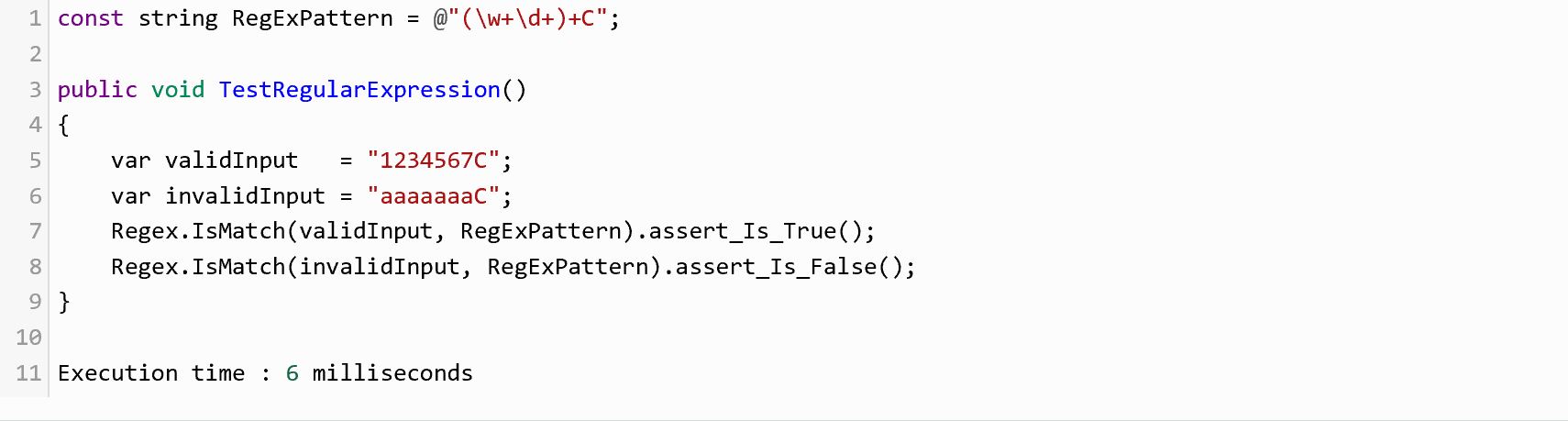
Ще разгледаме следния пример, който демонстрира това:

Регулярният израз, който използваме е *(\w+\d+)+C* . Това означава:

* Между един и неограничен брой пъти, колкото пъти е възможно
* \w+ Разпознава всеки символ измежду a-zA-Z0-9\_
* \d+ Разпознава цифрите от 0 до 9
* Разпознава символът С (има значение голяма и малка буква)

Така положителни съвпадения са 12С, 1231231232322С и отрицателни са !!!!!!С, аааааааС и ababababaC.

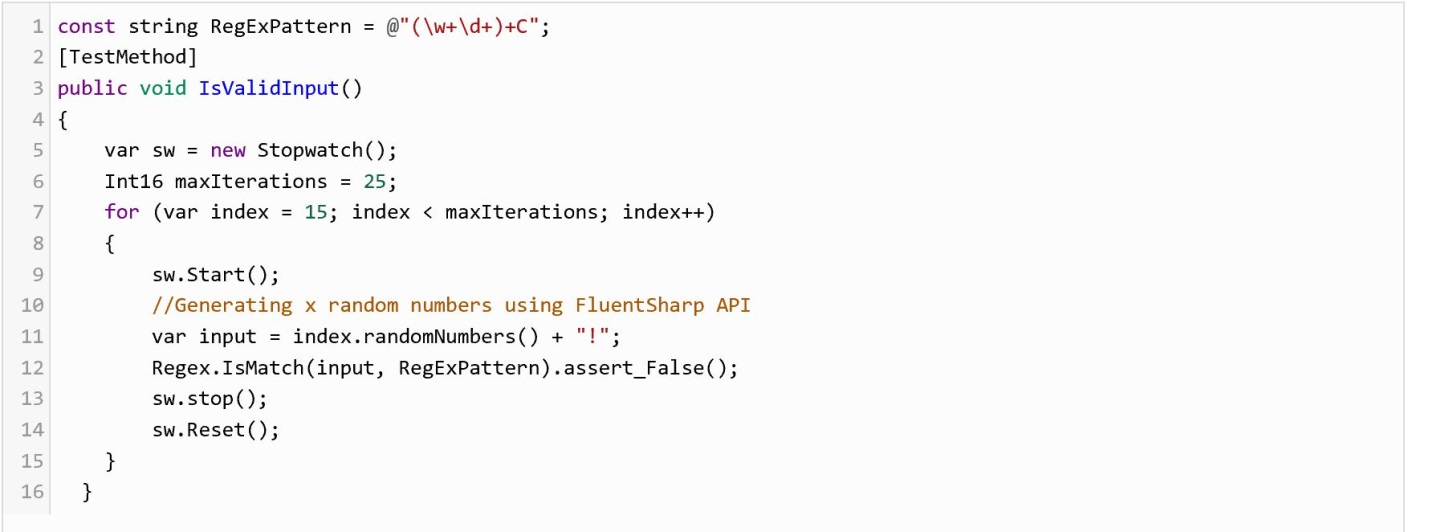
Следната фигура представлява код от Unit Test, който показва положително и отрицателно съвпадение.



Фиг. 2 Unit Test, демонстриращ положително и отрицателно съвпадение по даден регулярен израз и вход

Вижда се от измереното време, че при такъв малък вход, както и положителното, така и отрицателното съвпадение се откриват много бързо.

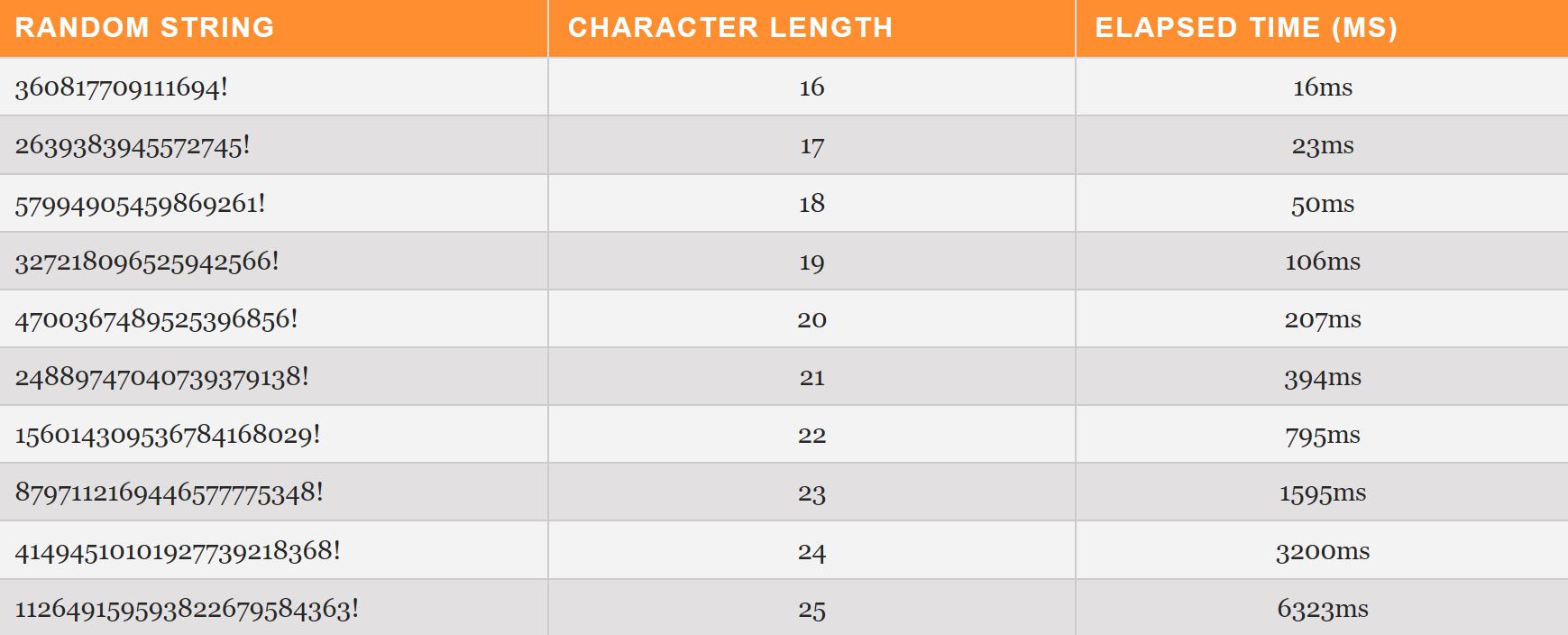
На фигура 3 е показан друг Unit Test, който демонстрира проблема с backtracking-а. Както се очаква, колкото по-дълъг е входния текст, толкова повече време отнема на алгоритъма да реши задачата. Кодът генерира 10 символни низа на случаен принцип, започвайки с дължина 15 на низа и увеличава дължината докато достигне 25 символа.



Фиг. 3 Unit test, който демонстрира как разликата в дължината на входа указва влияние

След пускането на теста е направена извадка от резултатите като е засечено времето за изпълнение на всяка итерация. Резултатите са сортирани от низа с най-малка дължина към тези с най-голяма. На Фиг. 4 се виждат конкретните стойности, които са измерени.

/Тестовете са проведени на машина със следните характеристики: Windows 7 32-bit operating System, 5.22 GB RAM. Intel(R) Core (TM) iT-3820QM CPU @2.7 GHz. /



Фиг. 4 Времената за разпознаване на 10 символни низа с различна дължина

Гледайки таблицата на фиг. 4 лесно можем да видим промяната във времето за изпълнение на различните входове срещу регулярния израз. Забелязваме, че времето расте експоненциално с нарастването на входа. Примерът ни показа, че при по-голям вход и без подходящи валидации на данните последствията за работата на нашето приложение са значителни.

След като показахме с прости примери как работи алгоритъмът и колко време му отнема за различен вход ще дмонстрираме проблем от реалния свят, с който се срещат уеб програмистите. Често срещана задача в уеб приложенията се състои от изискване на e-mail адрес от потребителя, за да се регистрира в програмата или да извърши друго действие. Модерните браузъри поддържат възможността за изкарване на съобщения разбираеми за потребителя, които указват, че въведения текст не съответства на формата, в който се очаква да е един e-mail адрес. Тези бързи валидации са бързи и приятни от потребителска гледна точка, но не може да се разчита само на тях. Злонамерен хакер , който „подслушва“, може да направи HTTP заявка и без да използва браузъра като използва прокси за прихващане на данните докато се движат по мрежата, да зареди заявката с подхотвен вход и да компрометира цялата система.

Следният пример използва валидация на входа, която се извършва на сървъра, използвайки регулярни изрази. Тества се не само това, дали регулярният израз валидира входа, но и как се държи, когато получи злонамерен вход.

За целта на теста взимаме „злия“ регекс **^(0-9a-zA-Z@([0-9a-zA-Z][-\w][0-9a-zA-Z].)+[a-zA-Z]{2,9})$,** който да валидира получения вход.

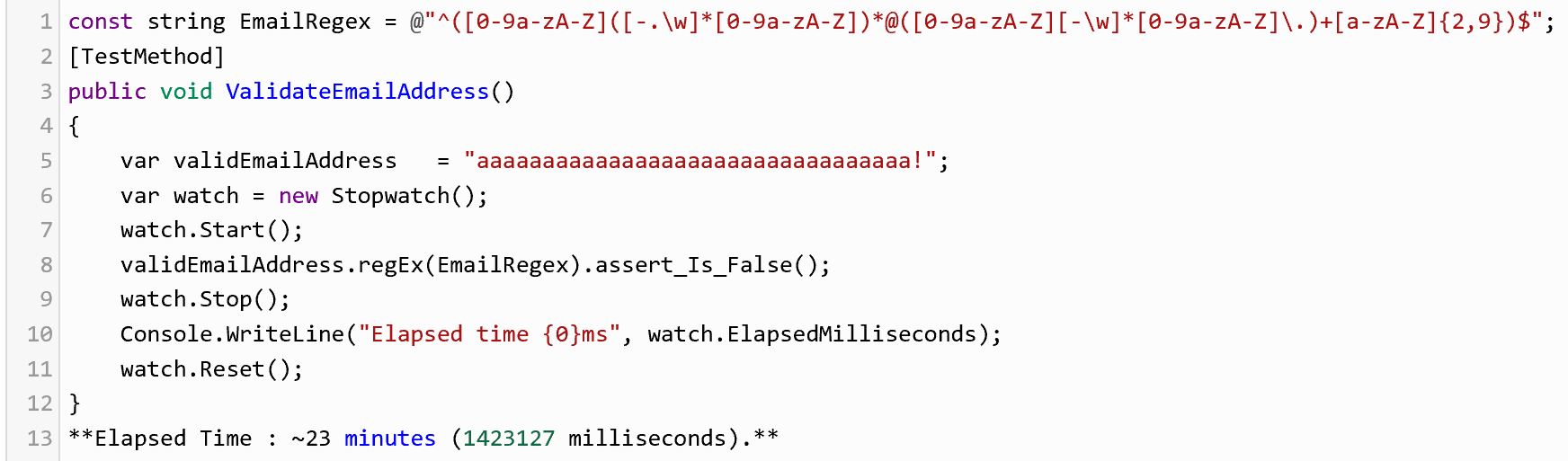
Със следния код, показан на фиг. 5 се уверяваме, че валидните и невалидните адреси се разпознават конретно от регулярния израз, което е ключово от програмна гледна точка.



Фиг. 5 Код демонстриращ как „зъл“ регекс разпонава коретно входа

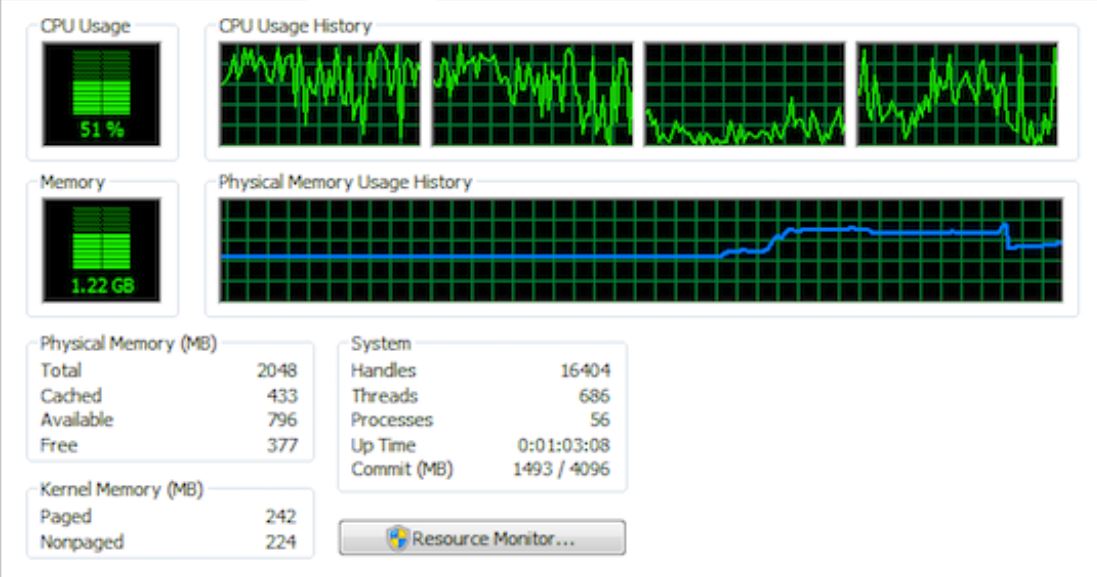
След като това работи и при валидни адреси и при невалидни, човек може да се подведе, че това е напълно достатъчно и да не забележи как „злият“ регекс може да компрометира системата му.

Въпреки, че от предния пример разполагаме с код, който правилно валидира входа трябва да разгледаме случая, в който получаваме произволен или злонамерен вход и да разберем, дали системата ни е изложена на риск. Трябва да проверим, дали кодът, който сме написали е достатъчно добър, за да ни защити от атаки. Примерът на фиг. 6 демонстрира това.



Фиг. 6 Демонстрация на атака върху регулярен израз.

Резултатът, който се наблюдава е показателен за това колко е лесно регекса, който сме използвали да бъде атакуван и приложението ни да спре да работи. За компютър с хардуерни характеристики като гореописаните 23 минути за обработка на този на пръв поглед елементарен вход ни показва, колко сме уязвими, когато използваме регулярни изрази и колко трябва да внимаваме при избора на такъв. Фиг. 7 ни показва как се справя компютърът с изпълнението на Unit test-а и какви ресурси използва.



Фиг. 7 Ресурсите, които използва компютърът при изпълнението на Unit test, атакуващ регулярен израз.

След като е толкова лесно системата да „увисне“ програмистът трябва да се увери, че когато използва регулярни изрази е направил всичко възможно, за да се предпази от подобна атака. В следващите няколко параграфа ще разгледаме начини за предвиждане на подобен проблем и решаването му.

В предишния пример открихме, че даден входен текст може да доведе до ReDoS атака и да спре работата на нашето приложение, при това използвайки не особено голям вход, в нашия случай беше 34 знака, но и с по-малко резултатът ще е подобен. Подходът за предвиждане на тази ситуация и прогнозирането й се базира върху изчерпателно тестване, за да могат да бъдат изпробвани всички възможности за един потребител да компрометира системата. Терминът за този подход се нарича Fuzzing. Тобаяс Клайн в книгата си A Bug Hunter's Diary A Guide Tour Throught the Wilds of Sofware security дефинира терминът по следния начин:

„Един съвсем различен подход при откриване на бъгове е познат като Fuzzing. Това е техника за динамичен анализ, която се състои от тестване на приложението с деформиран или неочакван вход. Не е лесно да се открие мястото в приложението, което е уязвимо, но сложният софтуер често не успява да сработи при деформирани входни данни.“

Мано Пол, написал книгата Official (ISC)2 Guide To the CSSLP  описва термина fuzzing по следния начин:

„Познат и като fuzz тест или инжектиране на грешни данни, това е brute-force начин на тестване, при който грешни (случайни или псевдо-случайни данни) се инжектират в софтуера и се наблюдава как работи. Резултатите от тези тестове са показателни за ефективността на входните валидации.“

Имайки пред вид предните дефиниции ще имплементираме нов Unit test, който ще ни позволи да генерираме вход и да тестваме регулярния израз. Ще използвайки следния регулярен ираз за валидация на e-mail [***\w-.]{1,}\@([\w]{1,}.){1,}[a-z]{2,4}$***. След провеждане на подробни тестове ще видим дали не сме изложени на атака от тип отказ на услуга. За да смятаме, че сме успели да се защитим от атака трябва по времето за изпълнение на всеки тест времето да е под 3 секунди.



Фиг. 8 Демонстрация на подробно тестване на регулярен израз със случаен вход.

При гореописаните технически характеристики на машината, върху която е пуснат тестът от фиг. 8 тестът минава. Както се вижда от задаването на максимален брой итерации 10000 и във всяка една от тях кодът (index \* 5) виждаме, че най-големият възможен низ, който ще генерираме се състои от 49995 символа. Уверихме се, че дори и при много голям вход времето за проверка на входа отнема не повече от 3 секунди. Едно значително подобрение би било ако първо се прави проверка за дължината на e-mail адреса. По този начин лесно ще се избегне инжектиране на огромен вход.

# MatchTimeout параметър за превенция на „катастрофален backtracking”

При разработката на приложение, използвайки Microsoft .Net 4.5 и нагоре програмистът може да се възползва от преимуществата на имплементацията на метода IsMatch от класа Regex. Този метод дава възможност да се зададе максимално време за изпълнение. Този нов параметър се нарича matchTimeout и според Microsoft:

„Параметърът matchTimeout указва колко време методът, който проверява дали входът отговаря на зададения модел, ще търси преди да бъде спрян. Използването на такъв параметър предотвратява негативните страни на backtracking частта от алгоритъма и пречи на зациклянето на процеса, когато открива близки съвпадения. Ако не бъде открито съвпадение в рамките на този интервал, приложението хвърля RegexTimeoutException. matchTimeout параметърът затрива всички тайм-аут интервали по подразбиране, който домейна на приложението е поставило.“

Следващият пример показва регулярен израз, за който се знае, че може да доведе до ReDoS атака. Ще тестваме e-mail адрес, който преди генерираше значителни странични ефекти в работата на приложението. След това ще видим как можем да намалим негативното влияние поставяйки тайм-аут параметър.



Фиг. 9 Демонстрация на използване на тайм-аут параметър

Пускайки теста от фиг. 9 виждаме, че минава, което означава, че на backtracking механизмът му отнема повече от 5 секунди да реши задачата. В този пример, приложението хвърля RegexTimeoutException, така че ние разбираме, че е отнело повече от 5 секунди. В нормален случай се очаква валидацията да приключи за под една секунда, но когато това не стане параметърът, който сме поставили се грижи за това системата да не „увисне“. Този начин на предпазване от атака е много добър поради факта, че програмистът е предвидил проблема и е направил информиран избор, за това как да намали вероятността от атака тип отказ на услуга.

# ReDoS атаки върху .Net MVC

Следващите няколко параграфа разкриват уязвимост в платформата за разработване не уеб приложение на Microsoft .Net MVC. Проблемът е описан теоритично, след което е показано как с visual studio 2013 се създава уеб проект, заедно с бъга в него вграден в платформата и как може да бъде използван.

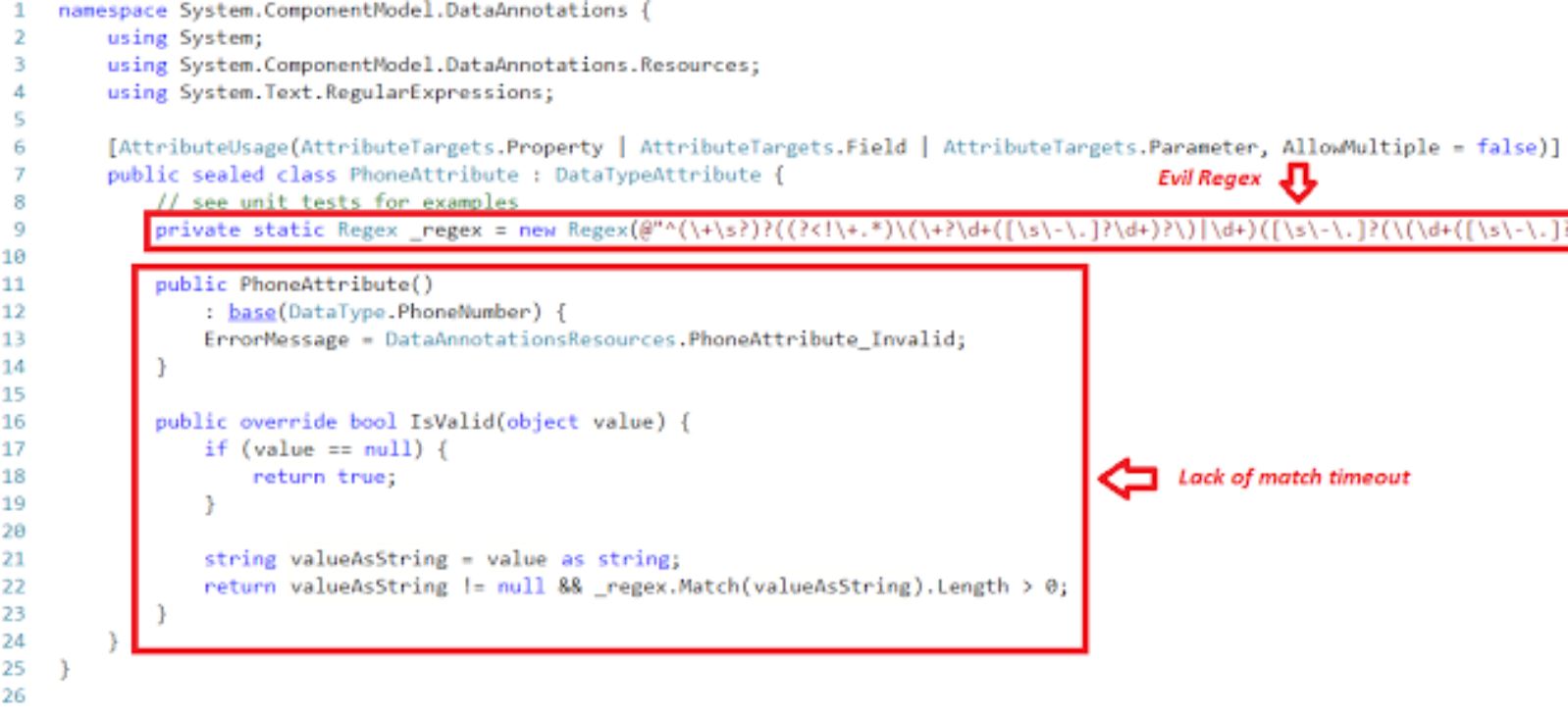
Платформата на Microsoft използва backtracking техниката, която беше описана по-горе и алгоритъмът за откриване на съвпадения е изграден върху математическото понятие недетерминистичен краен автомат, което също е описано в началото на доклада. Този подход се използва с цел възможност за валидиране на всички възможни входове, но опасността от използване на „зъл регекс“ и принуждаване на машината да смята експоненциален брой състояния остава.

В .Net Платформата (4.5) „зли“ регулярни изрази се изпозват по подразбиране в три класа – EmailAddressAttribute, PhoneAttribute, UrlAttribute, Които са част от библиотеката System.ComponentModel.DataAnnotations. Тези три класа извършват валидация на e-mail- адреси, телефонни номера и URL адреси в уеб формите. Това, което е най-притеснително е, че и трите не използват параметър за тайм аут.ET .NET Framework (4.5), Framework (4.5),

Следните три фигури (фиг. 10, фиг. 11, фиг. 12) показват слабите места в регексите на тези три класа, како и липсата на параметър, който би предотвратил увисването на системата.



Фиг. 10 Слаби места в кода на EmailAddressAttribute класа



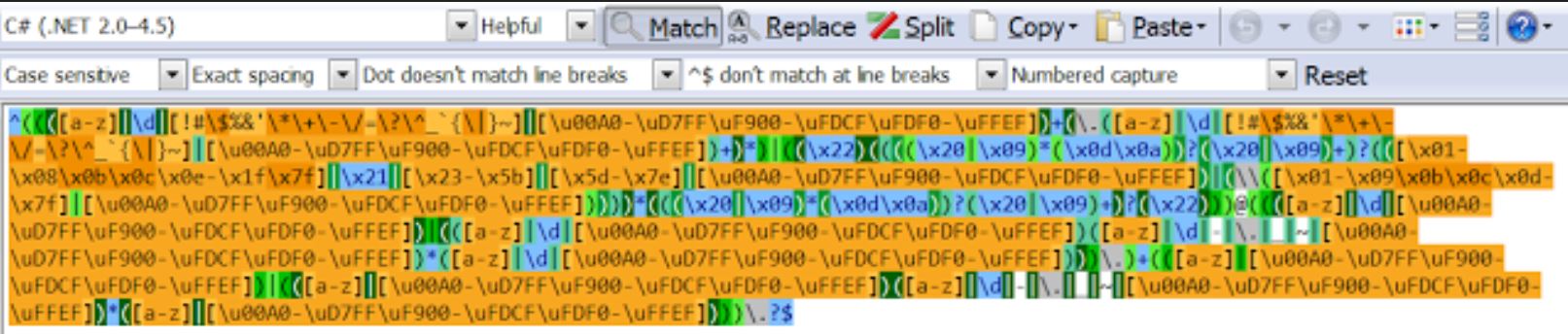
Фиг. 11 Слаби места в кода на PhoneAttribute класа

* 

Фиг. 12 Слаби места в кода на UrlAttribute класа

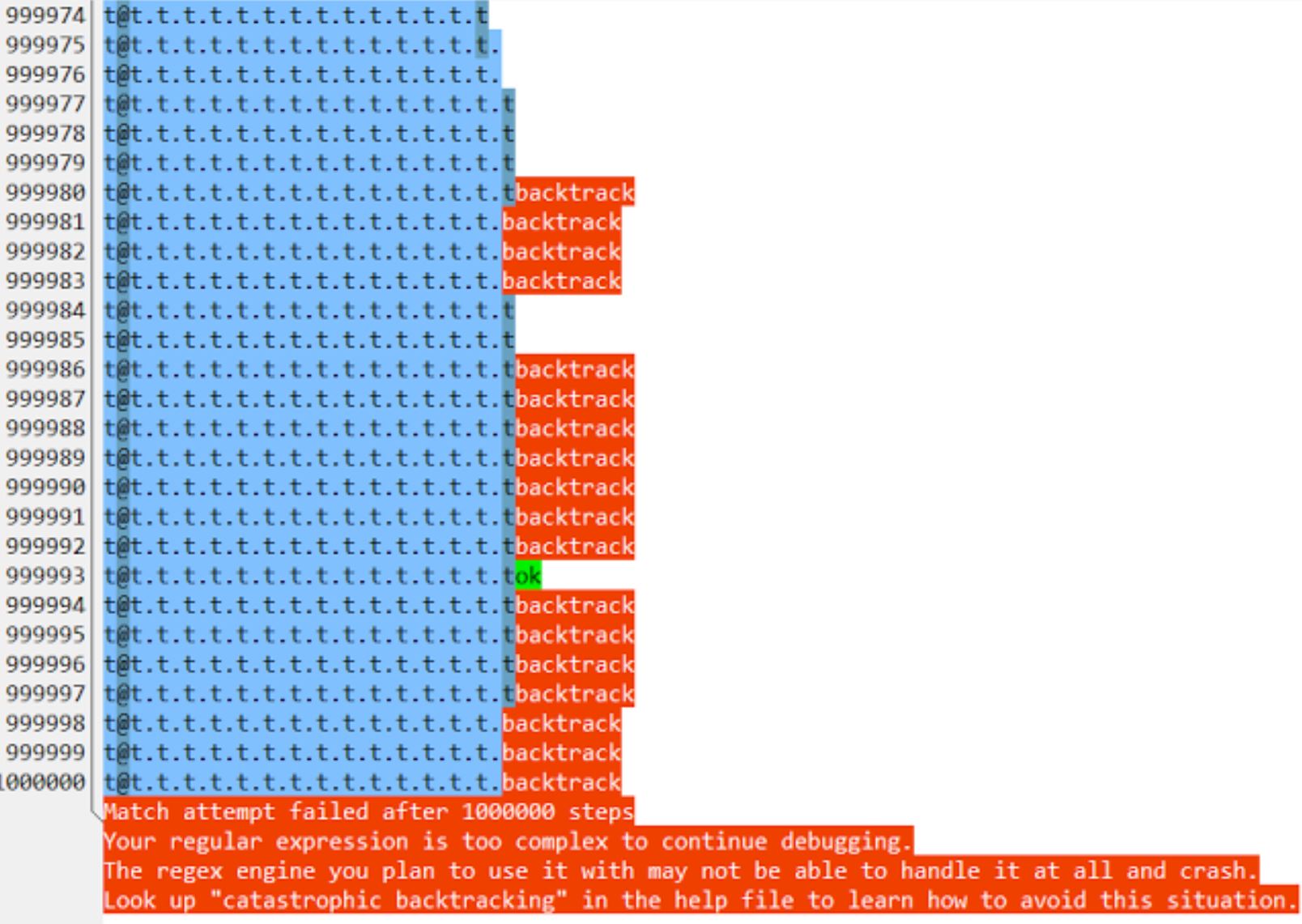
В резултат на тези слаби места атакуващият лесно може да създаде злонамерен вход, който да предизвика огромно количество изчисления и да причини атаката отказ на услуга върху конкретен контролер, например логин форма, която използва валидацията по подразбиране.

Отказа на услуга ще се окаже специфичен за конкретния контролер – логин формата, регистрационната форма, формата за контакти и др. В такъв случай потребителят евентуално ще може да се навигира в сайта, но няма да може да използва части от него. Причините регекса в класа за e-mail адрес да е „зъл“ са: голямата сложност, повторенията, високото ниво на влагане и рекурсията. Регексът е подложен на анализ от софтуера RegexBuddy специализиран в анализиране на регекси и резултатът е демонстриран на фиг. 13



Фиг. 13 Анализ на класа EmailAddressAttribute направен от софтуера RegexBuddy

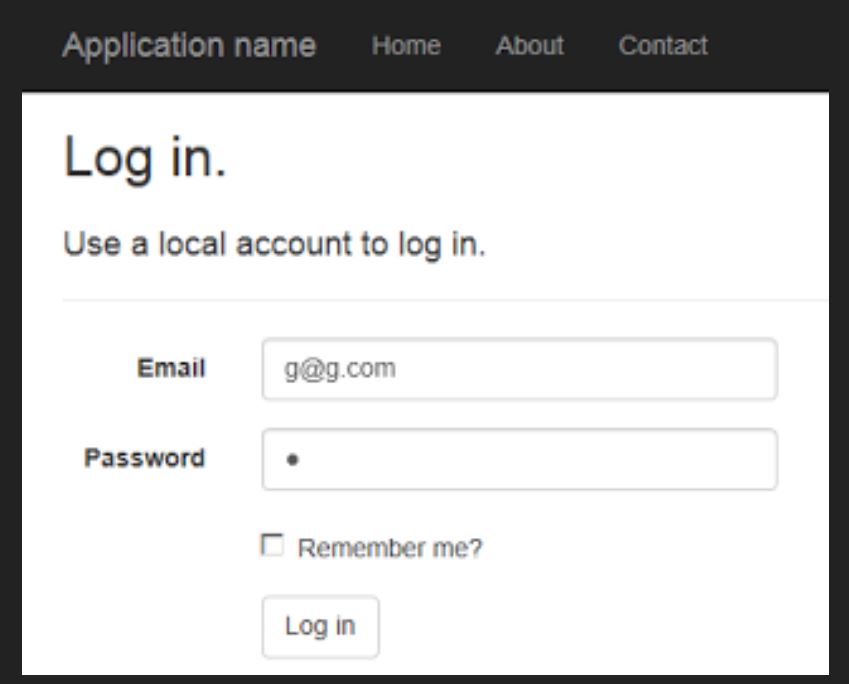
Катастрофалният backtracking е уловен от RegexBuddy и неговият вграден дебъгер (нагласен за C# - .Net 2.0 – 4.5) – с вход t@t.t.t.t.t.t.t.t.t.t.t.t.t.t.t%20, който никога няма да бъдe разпознaт от регулярния израз. Последните 26 операции могат да бъдат видяни на фиг. 14, където се вижда, че RegexBuddy е спрял изпълнението на програмата тъй като тя няма да приключи.



Фиг. 14 Прекратяване на изчисленията от RegexBuddy поради катастрофален backtracking

Този пример демонстрира катастрофалния backtracking. Софтуера спира изпълнението след 1 млн. Стъпки, но уязвимия клас EmailAddressAttribute, не поставя тайм аут атрибут и следователно .Net механизма за регулярни изрази ще продължава да прави изчислителни стъпки, което ще доведе процеса на w3wp.exe на уеб сървъра до 99% недостиг на CPU за известен период от време, например часове или дни в зависимост от подадения вход. Друг обеспокоителен факт е, че атакуващият може да нагласи скрипт, който автоматично да атакува уязвими форми на определен период от време.

Атаката се състои в изпращане на добре подготвена http post заявка към уеб форма използваща уязвимия клас EmailAddressAttribute. Примерът е демонстриран срещу .NET MVC уеб приложение, разработено с Visual Studio 2013. Приложението разполага с логин форма, която използва валидацията на e-mail адрес по подразбиране. По-долу е показан скрийншот от логин страницата (фиг. 15).



Фиг. 15 Скрийншот на логин формата от приложението, което ще бъде атакувано

Атакуващият може да приложи валидация на клиента като изпрати заявката чрез скрипт или прокси и да изманипулира заявката по следния начин:

*POST /Account/Login HTTP/1.1*

*Host: 192.168.0.13*

*User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:36.0) Gecko/20100101 Firefox/36.0*

*Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8*

*Accept-Language: en-US,en;q=0.5*

*Accept-Encoding: gzip, deflate*

*Referer: http://192.168.0.13/Account/Login*

*Cookie: \_\_RequestVerificationToken=FkLGrc6-XD2IBVU9g1nPycs0GTu3jWiK2QEyvR8IsowXAJU3C5fHlHvQvwGgB0VcN1FTa\_hB9KZ6Pi8SeI5EKpvz\_EeOqD7y\_FnipWJWqOU1*

*Connection: keep-alive*

*Content-Type: application/x-www-form-urlencoded*

*Content-Length: 239*

*\_\_RequestVerificationToken=HQq6--asc9wLbvnvuapMLuj5y9f8tSg9n0JiEFivqKv\_aeyl6eSaHaDtymjPgusP-spu-oUYa0xm7n\_RKjmS9WOU2so8S96X6oY7K2fd3lk1&Email=****t@t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.t.c%20****&Password=test&RememberMe=false*

В червено оцветеният текст е e-mail адреса, който атакуващият подава.

Кодът, който обработва заявката е показан по-долу:

AccountModelView.cs - use of [EmailAddress] default class in .NET

   public class LoginViewModel

    {

        [Required]

        [Display(Name = "Email")]

        [EmailAddress]

        public string Email { get; set; }

AccountController.cs – ModelState is validated when  the POST request occurs

// POST: /Account/Login

        [HttpPost]

        [AllowAnonymous]

        [ValidateAntiForgeryToken]

        public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)

        {

            if (!ModelState.IsValid)

            {

                return View(model);

            }

            // This doesn't count login failures towards account lockout

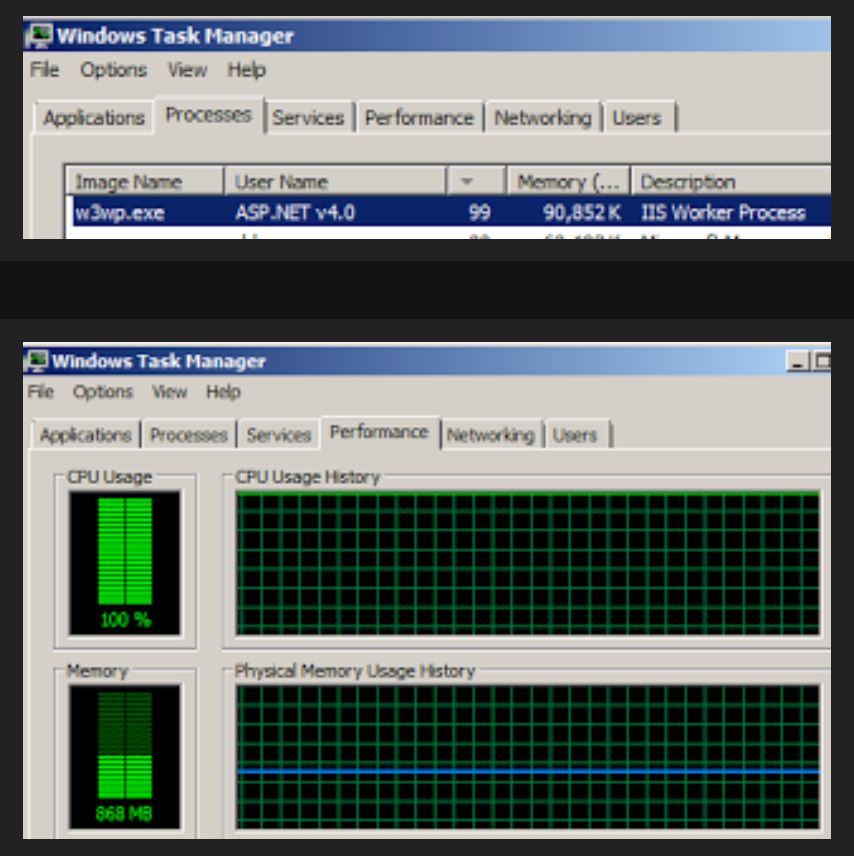
            // To enable password failures to trigger account lockout, change to shouldLockout: true

            var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, shouldLockout: false);

            switch (result)

            {

Резултатът от изпълнението на този код при получаване на гореописаната заявка е визуализиран на фиг. 16



Фиг. 16 Резултат от атаката чрез манипулирана заявка

Не са нужни специални обяснение на фиг. 16, за да разбере човек, че атаката срещу този регулярен израз е била фатална за приложението. Получавайки заявката, откзът на услуга настъпва за /Account/Login контролер класа. Като резултат от това никой потребител не може да използва /Account/Login контролер класа докато процеса на w3wp.exe е на 99% CPU недостиг. w3wp.exe процеса трябва да бъде спрян, за да възстанови приложението от атаката. След няколко ръчни възстановявания приложението става неизползваео и сървърът трябва да бъде рестартиран.

# Заключение за предпазване от атаки

За съжаление не съществува едно глобално решение за всички проблеми и в това число за проблемите, които възникват при използването на регулярни изрази. Няма как да сме напълно сигурни дали приложението ни не е изложено на риск и дали няма да бъдем атакувани в един момент. Това, което можем да направим обаче е да се опитаме да преценим колко сме уязвими използвайки unit тестове и да се защитим доколкото е възможно, пишейки максималко сигурен код.

Покривайки всеки ред код с unit тестове си гарантираме, че още на ранен етап сме успяли да засечем потенциални опасности и грешки. Най-добрата практика е създаването на множество такива тестове, които освен това целенасочено се опитват да атакуват и компрометират системата и то използвайки стандартни и не много трудни за имплементиране подходи като инжектиране на случаен вход, използване на техниката fuzzing, използване на комбинация от символи, надвишавайки допустимата дължина на текст и др.

# Източници

**[1]** https://www.owasp.org/index.php/Regular\_expression\_Denial\_of\_Service\_-\_ReDoS

**[2]** https://en.wikipedia.org/wiki/Regular\_expression

**[3]** https://dzone.com/articles/regular-expressions-denial

**[4]** http://blog.malerisch.net/2015/09/net-mvc-redos-denial-of-service-vulnerability-cve-2015-2526.html