Софийски университет “Свети Климент Охридски”

Факултет по математика и информатика

# HEIST – криптираната http информация може да бъде откраднатa през TCP прозорци

*реферат*

*по “Основи на сигурното уеб програмиране“*

*на Калина Стоянова Бухлева,*

*ф.номер 61681*

# Съдържание

Въведение....3
HEIST....4
HEIST от вътре....4
Определяне на точната дължина на отговора....7
HEIST и големи отговори....8
Атаки....9
CRIME....9
TIME....9
BREACH....9
Кражба на лична информация....10
Защита....10
Браузър....11
Забрана на „бисквитки“ от 3-та страна....11
HTTP слой....11
Спиране на компресията....11
Заключение....11
Използвани ресурси:....11

С HEIST атака може да откраднат лична информация, както и пароли и номера на кредитни карти от HTTPS-криптиран сайт. Все повече организации вярват, че самото използване на HTTPS спира множество хакерски атаки, затова и редица банки, доставчици на емайл услуги използват HTTPS. Миналото лято на Black Hat Конференция в САЩ беше показано нещо ново.Наречена HEIST новата техника атакува SSL/TLS и различни сигурни канали, за да открадне информация и по важното, това става през браузъра.

Двама белгийци, Mathy Vanhoef и Tom Van Goethem стоят зад новата атака представена през 2016 в Лас Вегас. Атаката не е особено новаторска, но е пореден знак, че timing атаките са се променили и стават все опасни.

HEIST изплозва слабостите на браузъра и лежащите отдолу HTTP, SSL/TLS и TCP слоеве. По-точно, става въпрос за страничен канал, който прихваща точния размер на cross-origin отговор. Този канал се възползва от начина по който се изпращат съобщения от TCP слоят и факта, че SSL/TLS не крие дължината на съобщението. Това значи, че атаки свързани с компресията като CRIME и BREACH стават все по-лесни за изпълнение. Още повече те показват, още няколко атаки използващи размера на изпратените съобщения.

# Въведение

Почти целия уеб трафик е криптиран, за нещастие това не значи нищо. Макар да стана тенденция да се откриват проблеми в SSL/TLS криптирането, никой не се възползва от тях. От гледна точка на хакера е много работа да следиш трафика между клиента и сървъра. Естествено това не спира тези, които имат достъп до незащитена мрежа. Какво обаче става, ако махнем това изискване. Например единственото нещо, което трябва да стане, за да хакера има достъп до комуникацията между клиент и сървър е клиента да изпълни JavaScript код. С HEIST ще ви покажа, че това е напълно възможно: физическата мрежа вече не е необходимост, за да направиш уеб атака.

HEIST е базирана на BREACH-атака представена на същата конференция преди три години. Разликата вместо MITM се ползва JavaScript код вкаран на сайта чрез XSS или XSRF. Този JavaScript код измерва времето за отговор на TCP. Чрез математически метод базиран на дължината на TCP прозореца откриват дали даден символ присъства в отговора. Спомага факта и, че SSL/TLS не крие дължината на съобщенията (слабост добре позната още от 1996 [2]). Засега BREACH атаката беше само теоритична или поне нямаше инструменти, които да е улесняват с появата на HEIST изглежда нещата ще се променят, но това все още е рано да се каже.

# HEIST

Да започнем с какво точно е HEIST и постепенно ще преминем към начините по който може да бъде ползвана.

## HEIST от вътре

От гледна точка на функционалност Fetch е подобно на XMLHttpRequest като и двете позволяват на разработчика лесно да прави заявки до сървъра. Основна разлика е възможността да fetch() -неш всеки ресурс включително уторизираните cross-origin заявки, до които по принцип нямаме достъп заради Same-Origin Policy приниципа.

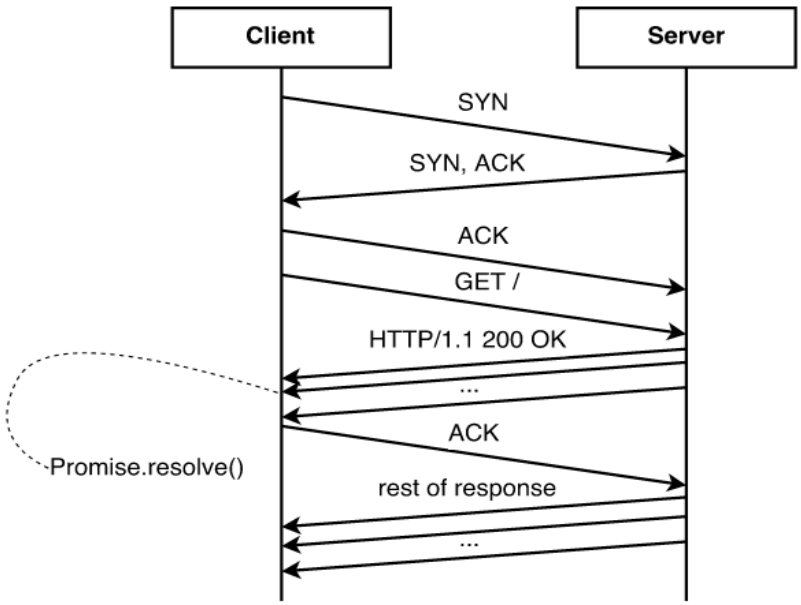
Още нещо, което трябва да се каже е, че Fetch работи с Promises вместо с Event-и. Това значи, че при получаване на отговор веднага се създава Promise обект. Promise обекта или носи отговора или съобщение за грешка взависимост от това дали е успяло да изтегли целия отговор. Интересното при изпълнението на Promise, е че става веднага щом първия байт от отговора е получен. Конкретно това значи, че след първия TCP handshake и SSL/TLS преговорите относно алгоритъма, браузъра изпраща GET или POST заявка до сървъра и чака отговор. Щом първият байт от този отговор достигне браузъра, Promise се изпълнява, страницата може да почне да се рендерира докато отговора още се тегли. Работи горе-долу така:

```
fetch('https://example.com/foo').then(function(response) {

        // first byte of ` response` received!

}, 1);
```

На пръв поглед, това поведение не изглежда опасно, и всъщност подобрява изпълнението на браузъра след като той може да започне работа по отговора получен от сървъра преди още да е изтеглил целия отговор. Поглеждайки по внимателно към TCP и добавяйки Resource Timing нещата почват да изглеждат тревожно. Нека първо видим какво става на TCP ниво на една обикновена HTTP заявка. След three-way handshake, клиентът праща TCP пакет съдържащ заявката, който обикновено се състои от стотина байта. Щом TCP пакета достигне сървъра се генерира отговор, който се изяраща обратно на клиента. Когато отговора стане по-голям от максималния размер на сегмента (MSS) отговора ще бъде разделен на няколко сегмента. Тези сегменти ще бъдат изпратени на клиента според TCP Slow Start алгоритъма. На практика това значи, че определен брой TCP сегменти (обикновено 10) са изпратени [1]. За всеки получен от клиента пакет, прозореца се разширява позволявайки изпращането на по-голям брой сегменти.




*Фиг 1: Типична HTTP заявка [1]*

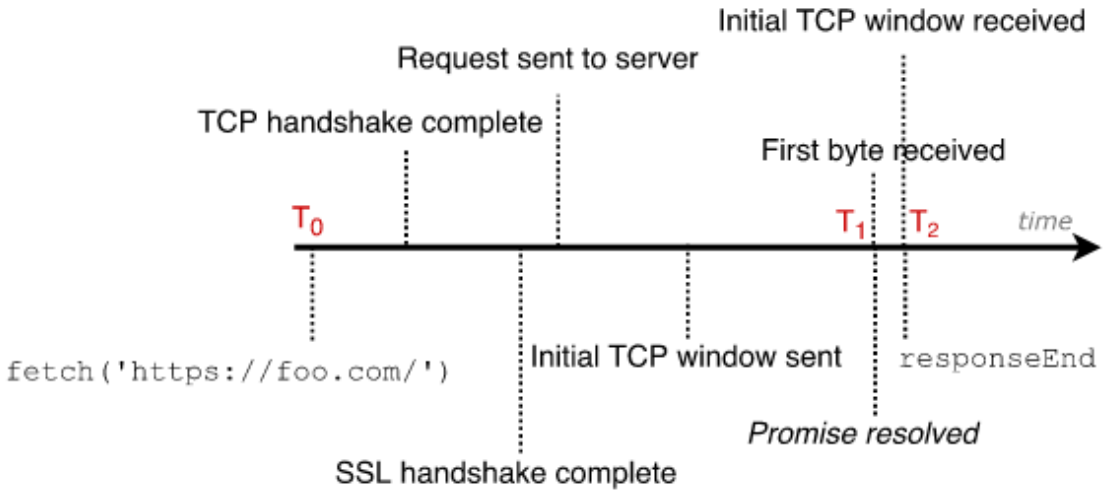
Връщайки се към Promise и fetch() функция можем да забележим, че времето по което Promise започва да се изпълнява съвпада с началото на първия TCP прозорец. Това значи, че ако знаем дали ресурса е свален ние знаем и дали той може да се побере в единствен TCP прозорец или му са нужни няколко. Затова ще се обърнем към Resource Timing APIв, чиято цел е точно това: показване на метрики свързани с това кога е инициализирана заявката и кога е завършила. Използвайки performance.getEntries() може да вземем PerformanceResourceTiming за съответната заявка и да получим времето за което отговора е бил свален с responseEnd атрибута. Следва пример как това би изглеждало с JavaScript код.

```
fetch('https://example.com/foo').then(function(response) {
        // first byte of `response` received!
        T1 = performance.now();
});
setInterval(function() {
        var entries = performance.getEntries();
        var lastEntry = entries[entries.length - 1];
        if (lastEntry.name =='https://example.com/foo') {
                T2minT1 = lastEntry.responseEnd – T1;
```

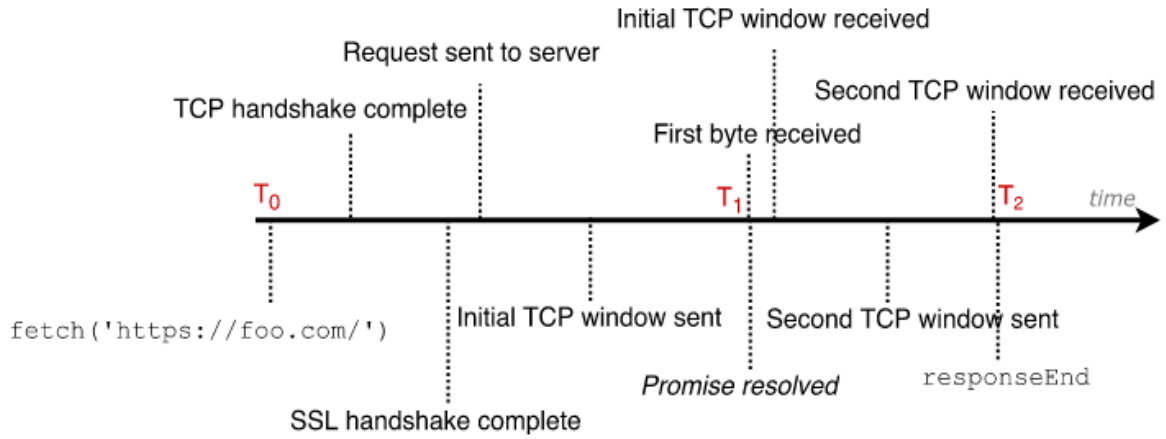
```
        }

    }, 1);
```

Можем да направим заявка в определен момент ($T_0$), да открием кога първия байт (и TCP прозореца) са получени ($T_1$), и кога целия отговор е получен ($T_2$). Гледайки на интервала между получаването на първия байт и кога ресурса е изцяло свален, можем да разберем дали отговора е отнел един или няколко прозореца. Фиг 2 показва времева линия на HTTP заявка и получения отговор от сървъра, като получения отговор се побира в един прозорец. Същото е показано и на следващата Фиг 3, но този път отговора отнема два прозореца. Гледайки фигурите е ясно, че когато само един прозорец е използван, $T_2 - T_1$ е много малко; на практика около 1ms. В случай, че втори TCP прозорец е нужен, $T_2 - T_1$ се увеличава и то чувствително.

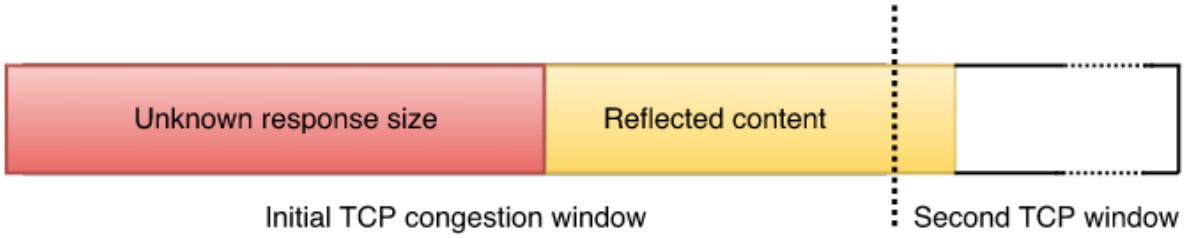



*Фиг 2*




*Фиг 3*

Да обобщим досега стана ясно, че лесно можем да определим дали дължината на отговора е по-дълга от началния прозорец. В повечето случаи това значи да определим дали отговора е повече от 14kB. Докато това може да бъде използвано, за да разберем дали потребителя е логнат на даден сайт (малко съобщение за грешка, когато не е и голямо




съобщение когато е), то това не е от голямо значение. Сега ще си поговорим за няколко техники с които HEIST става наистина опасен.

## Определяне на точната дължина на отговора

Естествено, много по-добре от това да знаем приблизително колко голям е отговора изпратен от сървъра, е да знаем истинския размер. Важно е да отбележа, че под точния размер се има предвид размера след gzip компресия и криптиране. Пример за атака знаейки размера на съобщението е BREACH. Атаката се възползва от начина на компресия на HTTP заявките за да разчете скритата информация. Ще обърнем внимание на атаката малко по-късно нека сега видим как да разберем точния размер на съобщението.

С първата техника ще покажем, че ако параметър от заявката ( GET или POST) участва в отговора, е достатъчно да разберем точния размер на съобщението. Можем да разделим съобщението на две части: частта, чиято дължина се опитваме да разберем и частта която вече знаем, защото е участвала в клиентското запитване. Последното естествено може лесно

да се контролира от хакера. За да разберем точния размер на съобщението ние може да подаваме многократно параметри с различна дължина опитвайки се да намерим най-големият параметър с който отговора се побира в един TCP прозорец. Можем да изчислим размера на непознатата част като извадим големината на първия TCP прозорец (фиксиран за всеки уеб сървър). След това просто изваждаме овърхеда от HTTP и SSL/TLS хедърите (и двете са предсказуеми). Пример: откриваме, че когато изпратения от нас параметър е с дължина 708 байта, се побира в точно един TCP прозорец; с 709 bytes, отговора се нуждае от 2 TCP прозореца. Имайки, че първоначалния TCP прозорец е 14600 bytes ( = 10 · MSS), 528 bytes за HTTP хедърите, 26 bytes – SSL/TLS овърхед; намираме, че размера на съобщението е 14600 − 529 − 26 − 708 = 13337 bytes.

Използвайки класически алгоритъм за бинарно търсене ще получим логаритмична скорост. Така разглеждайки 14600 възможни дължини ние правим само 14 заявки. Още може да подобрим алгоритъма. Първо браузъра позволява до 6 паралелни връзки за един хост. Използвайки всички 6 конкоретно, може да разделим пространството за търсене на 7 с всяка итерация. Така намаляваме времето нужно за изпълнение на атаката – с цената на повече заявки до сървъра. За да подобрим още атаката, лесно се забелязва, че най-голямата част от отговора е статична и предсказуема. Това ни позволява да намалим параметъра на търсене с около стотина байта.

Въпреки, че в повечето случаи изпратеният от клиента параметър участва в отговора има случаи в които не е така. Може да използваме по специфични атаки, когато изпратения параметър не участва в отговора.

## HEIST и големи отговори

В предишната част показахме как HEIST може да бъде използван, за да се получи точния размер на отговор, побиращ се в началния TCP прозорец. Докато това може да е достатъчно в определени случаи, определено не е приложимо във всички случаи. Нека да разгледаме ситуацията в която сървъра връща големи отговори. За това се връщаме към TCP Slow Start механизма. Както бе споменато алгоритъма кара TCP да започне с начално предположение за дължината на прозореца (обикновено: 10). За да позволи по-голям прозорец, TCP Slow Start предлага специален механизъм. По-точно за всеки пакет потвърден от клиента размера на прозореца се увеличава с 1.

Може систематизирано да увеличаваме TCP прозореца като първо изпратим заявка, чийто размер на отговора знаем. Например, ако пратим заявка с отговор побиращ се точно в 4 TCP пакета, и след това поискаме ресурса на който искаме да знаем размера, сървъра ще ни отговори с до 14 TCP пакета в началния прозорец. то плаващия прозорец беше разширен до 20, сървъра ще прати 15 пакета наведнъж.

Да обобщим: възможно е да се увеличи прозореца до определена дължина. При истинска атака, хакера би потърсил най-малкия брой TCP пакети, които биха се побрали в отговора. Това може да стане по същия начин като търсенето на подсказка с изпращането на

параметър, т.е. може да има логаритмична сложност. Щом този брой е намерен, може да се използва същата методология за намирането на размера. Интересното е, че увеличавайки размера на прозореца идва с още едно предимство. Използвайки този метод, дължината на съдържанието, което трябва да се повтаря трябва да бъде най-много 1 · MSS (1460 bytes). Това е полезно, когато параметър на GET заявка участва в отговора, след като някои сървъри ограничават максималната дължина на URL-а.

Комбинирайки всички описани дотук техники ни позволява да разберем точния размер на всеки ресурс стига параметъра от заявката да участва в отговора (или съществува друга алтернатива).

# Атаки

## CRIME

В доклад публикуван през 2002, John Kelsey разкрива, че компресията може да създаден страничен канал за изтичане на данни. Повечето алгоритми за компресия разчитат на повтарящи се модели в данните, за да може тези повторения да се кодират само веднъж. Това позволява появата на цял нов клас от атаки, разчитащи на познаването на символи в кодирана HTTP страница. Естествено тези атаки разчитат хакера да може да контролира съобщенията.

CRIME (Compression Ratio Info - leak Made Easy) атаката е създадена от Juliano Rizzo и Thai Duong през 2012. CRIME използва изтичането на информация от HTTP заявка ползваща SSL криптиране. Атаката изисква хакера да следи трафика и да направи множество заявки докато открие security cookie-то използвано от потребителя. Това може да стане като с всяка заявка изпраща параметър с различна дължина и наблюдава дължината на отговора. Ако дължината на отговора се промени, значи е познат символ. Това поведение се дължи на LZ77 частта от GZIP алгоритъма.

## TIME

TIME беше показана на Black Hat Europe конференция преди 4 години. Тя идва като подобрение върху CRIME. След като втората атака беше приложима само върху HTTP страници. Основната промяна идваща с TIME е, че хакера вече не се интересува от изпратените заявки, а от получените отговори. TIME атаката подобно на HEIST използваше само JavaScript, за да получи размера на отговора, но не получи широка популярност и бързо беше забравена. Всъщност HEIST може да се разглежда като подобрение на TIME атаката, тъй като отново тества дали отговора ще се побере в един прозорец.

## BREACH

BREACH атаката беше представена на същата конференция преди 4 години. Подобна на CRIME атаката BREACH се възползва от кеширането на данни и по-точно gzip компресията. За да изпълним оригиналната BREACH така, хакера трябва да може да:

- Вкара чист текст в отговора на сървъра.
- Следи трафика, за да измери размера на криптирания отговор.

Идеята зад атаката е, че хакера ще вкара префикс в токъна, gzip ще открие повторението и ще криптира по-добре получениия отговор. По време на демонстрацията беше показано, че получавайки отговора от сървъра ни трябват само 30-тина секунди да го декриптираме. Става толкова бързо, защото целта не е да се декриптира целия канал, а само да се извлече токъна.

С HEIST става много по-лесно да изпълним втората част от атаката: можем да намерим размера на изпратения отговор от сървъра използвайки само браузъра, вместо да правим MITM.

### Кражба на лична информация

Когато посещават сайт, потребителите обикновено имат страница показваща личните им данни, които те могат да споделят. Например newsfeed-а на facebook. Сега ще покажем, че чрез анализиране на отговори от уеб сайтове е възможно да видим информацията, която потребителя е споделил през сайта. Измислената компания (PatientWeb) сега има личен сайт, която позволява съхранение на здравни картони. Разработчиците на сайта са заинтересувани за неговото бързодействие и сигурност. Затова ще ползват HTTP/2 и ще се подсигурят, че BREACH не може да бъде ползван избягвайки да включват част от заявката в изпратения отговор.

Хакерът в този случай може да е трета страна, която продава здравна информация на застрахователни агенции.При посещението на заразен сайт, хакера първо проверява сайтовете в които потребителя  е влезнал, и пуска специфична атака зависимост от целта си. За PatientWeb, хакера може да ползва HEIST, за да определи заболяванията на клиента. В предварителната фаза се регистрират голям акаунти на PatientWeb и се избират различни заболявания за всеки от тях. На определени интервали размера (в байтове) на всеки newsfeed е събран.

За да определи състоянието на пациентите е да намери размера на newsfeed-а изпратен на потребителя.  За да се покаже точния размер на отговора, хакера може да комбинира различни ресурси (картинки, скриптове, ...), за да получи подобен отговор. Дори с краен списък от възможни заболявания, хакера разкодира отговора за по-малко от секунда.

## Защита

Атаките с HEIST разчитат на няколко мрежови слоя (браузър, HTTP, SSL/TLS, TCP). Ще разгледаме няколко възможни защити или поне начини за затрудняване на описаните дотук механизми.

## Браузър

### Забрана на „бисквитки“ от 3-та страна

Когато хакерът накара потребителя да направи заявка към определен сайт се връща специфичен отговор. Това е, защото всички „бисквитки“ на клиента участват в него, което значи, че за уеб сайта те са част от сесията на потребителя. Спирайки тези „бисквитки“ от заявките, те ще бъдат неавтентикирани и няма да получаваме желания отговор. Естествено не е възможно да се забранят всички „бисквитки“, но е възможно спирането им от трети места. По-точно това значи, че ако си на https://attacker.com , само бисквитки от този сайт ще бъдат включени в заяките. В момента повечето браузъри поддържат спирането на „бисквитки“ от непознати сайтове, но пък могат да спрат така функционалностите на други. Въпреки това няма по-добър начин да се избегне атаката.

## HTTP слой

### Спиране на компресията

HTTP отговорите обикновено се запазват използвайки компресия (или на SSL ниво, или използвайки gzip), за да се забърза комуникацията. Ясно е и, че това води до атаки CRIME (SSL компресия) и BREACH (gzip). За да се предпазим от тях може да стане и като забраним компресията. За съжаление това не ни предпазва от всички атаки свързани с размера на отговора.

# Заключение

Макар за сега HEIST да не се класифицира за опасна заплаха според експертите по сигурност, тя улеснява хакерите. Определянето и като заплаха, а не риск се дължи най-вече на липсата на популярност и това, че още никой не е написал скрипт, който масово да се използва. Важно е да се спомене, че още преди представянето на атаката експертите по сигурността бяха при Google и Microsoft, за да им съобщят за откритията си. За мен това е само началото на серия от атаки започващи от браузъра и тревожен знак, че начина по който пишем уеб трябва да се промени.

# Използвани ресурси:

[1] HEIST: HTTP Encrypted Information can be Stolen through TCP-windows, Mathy Vanhoef and Tom Van Goethem, Black Hat USA, 2016, 2016.
[2] Tal Be’ery and Amichai Shulman. A perfect crime? only time will tell. Black Hat Europe, 2013, 2013.
[3] Fetch Api
[4] HEIST Vulnerability – Overview and BIG-IP Mitigation
[5] Black Hat 2013 - SSL, Gone in 30 Seconds - A BREACH beyond CRIME