Статья Ransomware - все по взрослому или оптимизация работы (конкурс 2020)

xss.is/threads/34601

Ransomware - все по взрослому или оптимизация работы. merdock для конкурса XSS.is (ex DaMaGeLaB) 2020

1. Вступление
2. Оптимизация вычислительной нагрузки и алгоритма шифрования
3. Оптимизация работы с файловой системой
4. Ошибки в реализациях
5. Выводы
6. Исходники

1. Вступление

Статью пишу второй раз, первую Вы не найдете, так как первую случайно удалил. Данная статья создана на основе опыта и реверса Ransomware, он же шифровальщик (тип зловредного программного обеспечения, предназначен для вымогательства).

Решил поделиться методиками оптимизации данного типа зловреда и ошибками которые многие допускают при их написании. Я не буду разбирать эксплойты, т.к. дыры постоянно закрываются, рассмотрим ядро шифровальщика.
Статья и исходники создана сугубо в познавательных целях, основана на статистических данных собранными мной для исследования и обоснования.

2. Оптимизация вычислительной нагрузки и алгоритма шифрования
Наиболее хорошие результаты показал алгоритм шифрования AES(Advanced Encryption Standard, также известный как Rijndael), это симметричный алгоритм блочного шифрования.
Среди всех алгоритмов он показал наиболее высокую скорость и криптостойкость, мы будем использовать AES с ключом 256 бит с сцеплением блоков CBC, без использования PKCS7 padding (нам отступы не нужны, так как будут нарушать симметричность данных), без использования хеширования ключа алгоритмом SHA-256 (256 ключ будем сами генерировать и его длинна будет определена заранее), без использования сохранения вектора блоков (вектор рекомендую сохранять с ключом, чтобы усилить криптостойкость и не нарушать симметричность данных).
Чтобы усилить криптостойкость (тут паранойя) я предлагаю использовать гибридный метод, т.е. поверх шифрованных данных будем использовать быстрый симметричный алгоритм шифрования RC4 без блокового сцепления, ключ так же будем хранить вместе с вектором и ключом AES.
Данный подход даст возможность увеличить вариативность против атак подбора ключей и не сильно увеличить нагрузку. Код будет без использования сторонних библиотек, таких как OpenSSL или CryptAPI, это даст надежность (закладки и моя параноя) и независимость, а так же уменьшит шанс runtime детектов антивирусных программ.

Выбор AES был обучловнет по причине того что все современные процессоры поддерживают AES-NI (Intel Advanced Encryption Standard New Instructions) - расширение системы команд микропроцессоров, т.е. эта технология даст возможность без ущерба криптостойкости увеличить скорость шифрования и снизить нагрузку на процессор.

Я написал программу и выложил исходники для сбора статистики и проверки алгоритмов. Суть программы заключается в том чтобы продемонстрировать и показать наиболее эффективный метод шифрования. Методика теста заключается в создании массива случайных данных, их шифрование AES 256 с использованием AES-NI и без, с использованием RC4 и без, на разном количестве данных.
И так приступим.

На скрине мы видим сравнительный тест на длине данных 16кб (эта цифра выбрана не случайно в следующем разделе причины выбора будут объяснены)

x86 без AES-NI img1_86_not_aesni.png

x64 без AES-NI
img1_64_not_aesni.png

x86+AES-NI img1_86_aesni.png

x64+AES-NI img1_64_aesni.png

Так же я хотел добавить тесты с использованием GPU, и даже почти написал софт, но нашел статистику проведенную уже другим человеком, из нее видно что шифрование через CPU без AES-NI и чреез GPU на 16кб составляет небольшую раздницу.
img1_GPU_CPU_not_aesni.png

Если посмотреть мои тесты то использование CPU с AES-NI 16кб по скорости будет сравнительно с GPU, т.е. использование в шифровальщике GPU не имеет большого смысла. GPU выигрывает только при распараллеливании процессов шифрования файлов, но прирост при сохранении файлов будет потерян.

Выводы тестов показывают, что технология AES-NI дает серьезный прирост в скорости шифрования 3-5 раз, техника не имеет внешних зависимостей, так же на 16кб разница между AES и AES+RC4 не столь существенна.

3. Оптимизация работы с файловой системой

Самым узким местом в реализации шифровальщика становиться файловая система, т.е.

```
Platform: 86 bit
AES: 256bit
AES-NI: support, not use
Count array: 1024 items
Count data: 16 kb
Generate array: wait...ok
Encrypt AES array: wait...141 ms
Decrypt AES array: wait...140 ms
Test: ok
Encrypt AES+RC4 array: wait...187 ms
Decrypt AES+RC4 array: wait...219 ms
Test: ok
```

```
Platform: 64 bit
AES: 256bit
AES-NI: support, not use
Count array: 1024 items
Count data: 16 kb
Generate array: wait...ok
Encrypt AES array: wait...141 ms
Decrypt AES array: wait...140 ms
Test: ok
Encrypt AES+RC4 array: wait...219 ms
Decrypt AES+RC4 array: wait...202 ms
Test: ok
```

```
Platform: 86 bit
AES: 256bit
AES-NI: support, use
Count array: 1024 items
Count data: 16 kb
Generate array: wait...ok
Encrypt AES array: wait...47 ms
Decrypt AES array: wait...31 ms
Test: ok
Encrypt AES+RC4 array: wait...109 ms
Decrypt AES+RC4 array: wait...94 ms
Test: ok
```

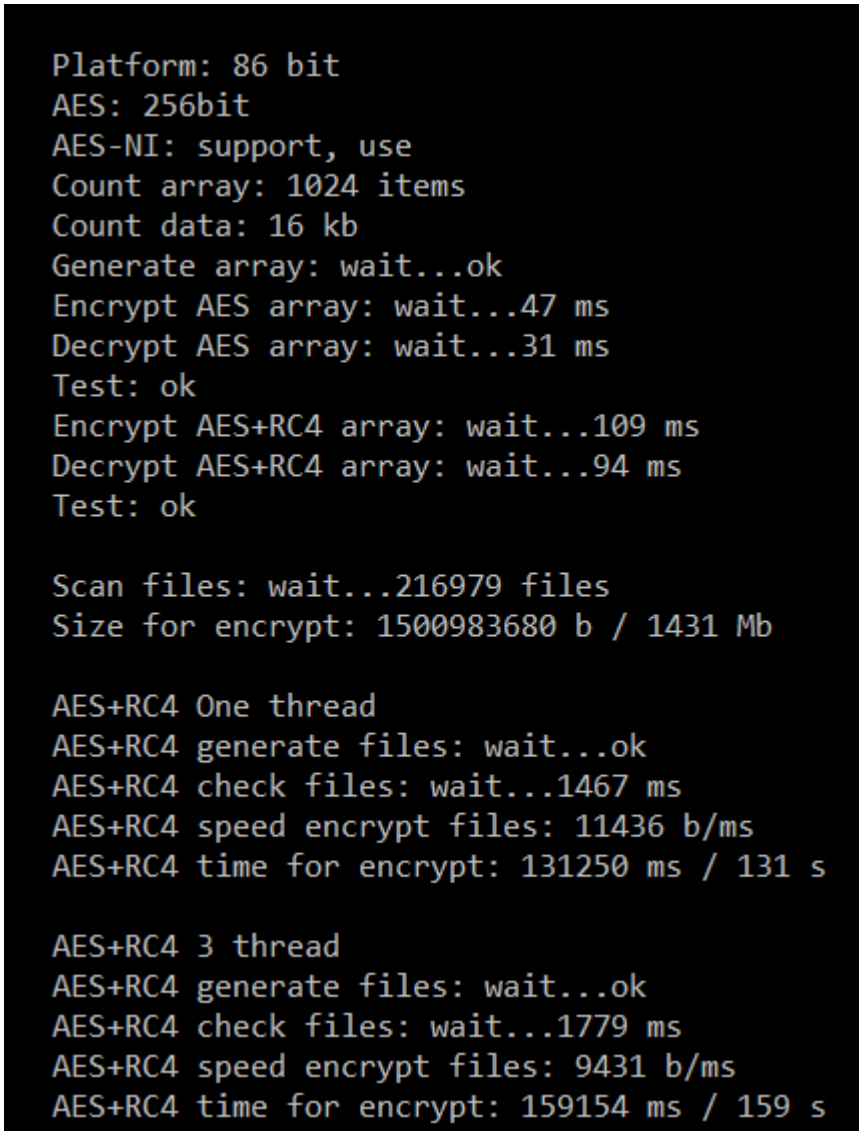
жесткий/HDD или твердотельный/SSD диск. Для оптимизации мы будем использовать частичное шифрование с частичной перезаписью файла, именно по этой причине нам был необходим полностью симметричный алгоритм сцеплением блоков, чтобы не менять длину файлов. Для данной техники мы не будем считывать весь файл и не будем его весь сохранять, только начальную часть в размере 16кб (этого достаточно чтобы любой файл потерял целостность и не смог был восстановлен, кроме архивов, их желательно шифровать первые 1-2мб).

```
Platform: 64 bit
AES: 256bit
AES-NI: support, use
Count array: 1024 items
Count data: 16 kb
Generate array: wait...ok
Encrypt AES array: wait...47 ms
Decrypt AES array: wait...16 ms
Test: ok
Encrypt AES+RC4 array: wait...109 ms
Decrypt AES+RC4 array: wait...94 ms
Test: ok
```

File Size (bytes)	Encryption			
	Time (seconds)		Throughput (bytes per second)	
	CPU	GPU	CPU	GPU
1202	0.000152	0.000732	7921052.63	1644808.74
4652	0.000689	0.000770	6769230.77	6057142.85
9302	0.001418	0.000784	6564174.89	11872448.97
18602	0.002545	0.000807	7313163.06	23063197.03
37202	0.004985	0.000886	7463189.57	41990970.65
74402	0.010099	0.001048	7367462.12	70996183.21
148802	0.020109	0.001302	7399870.70	114288786.48
297602	0.039651	0.001896	7505586.24	156964135.02
595202	0.079503	0.003145	7486560.26	189254054.05
1190402	0.158507	0.005316	7510103.65	223928517.68

Для ускорения работы стоит использовать много поточность поиска, т.е. на каждый диск один поток который ищет файлы и заносит их путь в буфер. Так же используем 16кб шифрования начальных данных файла без его полной перезаписи, 16кб потому что обычно стандартный размер кластера блока записи на диск 4кб, значит нам надо использовать кратную величину.
В тестах на скине мы видим тест скорости работы шифровальщика для заданного диска в одном потоке и в 3х потоках, для SSD и для HDD на реальных носителях в одной системе.

x86 HDD img2_86_hdd.png



x64 HDD img2_64_hdd.png

```
Platform: 64 bit
AES: 256bit
AES-NI: support, use
Count array: 1024 items
Count data: 16 kb
Generate array: wait...ok
Encrypt AES array: wait...47 ms
Decrypt AES array: wait...16 ms
Test: ok
Encrypt AES+RC4 array: wait...109 ms
Decrypt AES+RC4 array: wait...109 ms
Test: ok

Scan files: wait...216978 files
Size for encrypt: 1500967296 b / 1431 Mb

AES+RC4 One thread
AES+RC4 generate files: wait...ok
AES+RC4 check files: wait...1404 ms
AES+RC4 speed encrypt files: 11950 b/ms
AES+RC4 time for encrypt: 125603 ms / 125 s

AES+RC4 3 thread
AES+RC4 generate files: wait...ok
AES+RC4 check files: wait...1435 ms
AES+RC4 speed encrypt files: 11691 b/ms
AES+RC4 time for encrypt: 128386 ms / 128 s
```

x86 SSD img2_86_ssd.png

x64 SSD img2_64_ssd.png

```
Platform: 64 bit
AES: 256bit
AES-NI: support, use
Count array: 1024 items
Count data: 16 kb
Generate array: wait...ok
Encrypt AES array: wait...63 ms
Decrypt AES array: wait...15 ms
Test: ok
Encrypt AES+RC4 array: wait...110 ms
Decrypt AES+RC4 array: wait...109 ms
Test: ok

Scan files: wait...216977 files
Size for encrypt: 1500950912 b / 1431 Mb

AES+RC4 One thread
AES+RC4 generate files: wait...ok
AES+RC4 check files: wait...2496 ms
AES+RC4 speed encrypt files: 6722 b/ms
AES+RC4 time for encrypt: 223289 ms / 223 s

AES+RC4 3 thread
AES+RC4 generate files: wait...ok
AES+RC4 check files: wait...2480 ms
AES+RC4 speed encrypt files: 6765 b/ms
AES+RC4 time for encrypt: 221870 ms / 221 s
```

На скинах где используем один поток шифрования и записи данные почти не отличаются от того где используемым три потока.
Как видно из тестов вышее скорость работы с файловой системой при увеличении потоков не сильно отличается (%5 на каждый дополнительный поток, но не более 5), т.е. мы упираемся в ограничения записи (что и следовало ожидать).
Вывод показывают, что нет смысла делать кучу потоков. Оптимальный вариант на каждый диск по потоку поиска и один поток на весь буфер шифрования.

4. Часто встречающиеся ошибки в реализациях

Частые ошибки в реализации шифровальщиков:
- читают, шифруют и записывают файл полностью
- используют недостаточно кислотостойкие алгоритмы
- при поиске не учитывают папки-ссылки и не ограничивают вложенность рекурсии при поиске
- не учитывают юникод в названиях файлов и папок

- не ограничивают буферы для получания полного пути (ос виндовс есть ограничения на длину)
- не используют несколько источников для отсылки ключа(если не оффлайн шифровальщик)
- не используют несколько источников для выкупа ключа (WannaCry использовал почту и ее заблокировали)
- часто используют библиотеки шифрования не учитывая криптостойкость
- не делают балансировку нагрузки (или слишком много потоков или слишком мало)

5. Выводы

Как видно из тестов если использовать предложенную методологию и шифровать ценные файлы, то скорость шифрования без потери криптостойкости одного среднестатистического компьютера может занимать десяток секунд.

Всем кто дочитал большое спасибо, голосуйте за меня и я сделаю вашу жизнь веселее!

6. Исходники

Использовалось Delphi XE10, будет работать и на других дельфи после мелких изменений

Last edited: Jan 26, 2020