Использование микроконтроллеров VaultIC в качестве электронных ключей для защиты программного обеспечения от нелегального копирования

Защита программного обеспечения от нелегального копирования и использования — головная боль для многих разработчиков коммерческих продуктов. Придумано огромное количество методов борьбы с пиратством, однако все их можно условно разделить на две большие группы: программные и аппаратные методы. В последнее время большую популярность среди аппаратных методов получил способ организации защиты с помощью специальных внешних устройств — электронных ключей, или токенов. Главной особенностью таких устройств является аппаратная защита от различных атак (по сторонним каналам, реверс-инжиниринга и т. п.), а также наличие специальных сопроцессоров для ускорения выполнения криптографических операций. Благодаря этому токены могу применяться в качестве ключевого звена защиты программного обеспечения, так как их взлом или копирование являются крайне трудоемкой задачей, которая, в конечном счете, нецелесообразна для потенциального злоумышленника.

Алексей САБАДАШ
sai@efo.ru

На рынке существует немало токенов, специально предназначенных для защиты ПО. Они имеют разную степень надежности, используют различные методики обеспечения защиты, но их объединяет одно: каждый производитель имеет свой собственный SDK (Software Development Kit) — библиотеки, API-вызовы, определенные заранее схемы защиты. Естественно, находятся люди, обладающие специальными навыками, которые способны с помощью реверс-инжиниринга понять принцип функционирования такой унифицированной схемы защиты для каждой модели токена, а впоследствии придумать, как ее обойти. На специализированных форумах распространяются утилиты, способные «отвязать» любую защищенную программу от токена определенной модели. Это разного рода распаковщики, эмуляторы и т. п. Причем чем популярнее модель токена, тем проще найти такую утилиту или же человека, способного снять защиту в индивидуальном порядке.

В этой статье мы рассмотрим, как можно организовать адекватную защиту программ с помощью защищенного микроконтроллера общего назначения — микросхемы VaultIC460 от компании Inside Secure. Данный микроконтроллер интересен тем, что он имеет предустановленную прошивку: все распространенные криптографические алгоритмы (шифрование, цифровая подпись, хэширование) уже реализованы на кристалле. Соответственно, от разработчика требуется только использовать их в своем приложении для персонального компьютера. В частности, микроконтроллер можно использовать и для создания собственной реализации механизма защиты ПО. Этот подход имеет ряд преимуществ относительно использования готовых популярных решений. Во-первых, такая схема будет уникальной, и потенциальному взломщику понадобится «с нуля» исследовать механизм привязки за неимением готовых наработок. Во-вторых, использование VaultIC экономически более выгодно. Это особенно важно, когда стоимость лицензии программы сопоставима со стоимостью токена. Вряд ли потенциальные потребители программного продукта обрадуются увеличению его цены, скажем, в два раза — без расширения его функционала, а только из-за того, что разработчикам захотелось внедрить новое средство борьбы с пиратами. С другой стороны, может показаться, что такой способ потребует гораздо больше временных ресурсов, чем в случае использования готовых решений защиты. Мы же постараемся показать, что добавление базовой защиты, которая отпугнет подавляющее большинство «крэкеров», с помощью VaultIC не такая уж и сложная задача, не требующая каких-то специальных навыков.

Схемы защиты

Существует несколько подходов к организации защиты ПО с помощью электронных ключей. В данной статье мы продемонстрируем реализацию двух из них: проверка наличия подключенного токена и шифрование критичных областей программы с помощью токена. Такой выбор технологий защиты обусловлен тем, что на их примере легко показать возможности VaultIC. Кроме того, оставшиеся популярные методы либо не обеспечивают должного уровня надежности, либо их реализация неосуществима с использованием VaultIC. Так, например, шифрование программы полностью, конечно, способно обеспечить защиту от статического анализа кода, однако в момент выполнения программа в расшифрованном виде будет находиться в оперативной памяти компьютера, поэтому

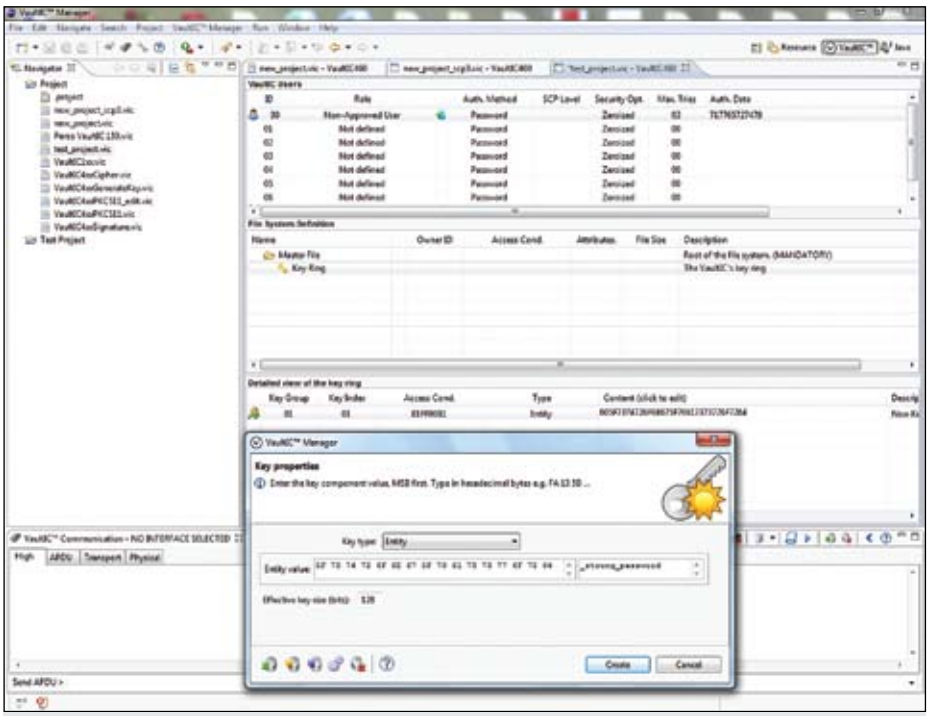

Рис. 1. Создание ключа в программе VaultIC Manager

сделать дамп полностью рабочей программы для взломщика не составит никакого труда. Другой набирающий в последнее время популярность метод — выполнение части кода непосредственно на самом электронном ключе — не может использоваться из-за наличия на кристалле VaultIC предустановленной прошивки, которую невозможно изменить. Этот метод хорошо зарекомендовал себя, однако применить его к конкретному программному продукту не всегда возможно: набор «понимаемых» микроконтроллером инструкций серьезно ограничен, следовательно, далеко не каждый алгоритм можно на него портировать. Поэтому многие разработчики по-прежнему отдают предпочтение классическим методам защиты. При их комбинировании, рациональном использовании и дополнении программными методами (например, обфускация, антиотладочные методы и др.) можно добиться уровня защиты, сравнимого с вынесением части кода программы в токен.

Для демонстрации реализации схем защиты с помощью VaultIC в качестве защищаемой программы мы воспользуемся свободным текстовым редактором metapad, распространяемым с исходными кодами. В роли защищаемой функции будет служить сохранение текстового файла, набранного или отредактированного в metapad, на жесткий диск — функция ***SaveFile()***.

Схема 1. Проверка наличия токена

Схема защиты с помощью проверки наличия токена — по-прежнему наиболее популярная. Она заключается в том, что в одном или нескольких местах программы происходит обращение к токену (например, считывается его уникальный идентификационный номер) и в случае его недоступности программа заканчивает работу или происходит выход из критичной функции. Рассмотрим реализацию этой схемы на примере VaultIC. Сначала с помощью специальной утилиты VaultIC Manager создадим пользователя User0 с паролем qwerty и ключ типа «**Сущность**» (**Entity**) с параметром ***_strong_password***, а затем загрузим получившуюся конфигурацию в микроконтроллер (рис. 1).

Теперь внесем необходимые изменения в исходный код metapad. Во-первых, добавим функцию инициализации токена ***InitToken()***, выполняющую загрузку библиотеки для работы с VaultIC и авторизацию созданного нами пользователя, а также функцию ***ReleaseToken()***, которая завершает сеанс пользователя и освобождает библиотеку.

```
int InitToken (void)
{
   …

   hMod = OpenLibrary( LIB_PATH );

   …

   commsParams.Params.VltPcscInitParams.pu8ReaderString =
(VLT_PU8)"Inside Secure VaultIC 460 Smart Object 0";
   commsParams.u8CommsProtocol = VLT_ISO_T0_COMMS;
   if( VLT_OK != VltInitLibrary( &commsParams ) )
      {
   CloseLibrary( hMod );
      MessageBox(NULL,"Token not connected!","Error", MB_
ICONSTOP);
      return( VLT_INIT_LIB_FAILED );
      }
   …
   if ( VLT_OK != (usActualSW = theBaseApi->VltSubmitPassword(VLT_
USER0,
      VLT_NON_APPROVED_USER,
      6,
      (VLT_U8*)"qwerty") ) )
      {
      MessageBox(NULL,"User login failed!","Error", MB_ICONSTOP);
      return ( usActualSW );
      }
return 0;
}

int ReleaseToken(void)
{
   if (VLT_OK != ( theBaseApi->VltCancelAuthentication( ) ) )
   {
      MessageBox(NULL,"User logout failed!","Error", MB_ICONSTOP);
      CloseLibrary( hMod );
      return 1;
   }
   CloseLibrary( hMod );
   return 0;
}
```

Вызов ***InitToken()*** разместим в начале основной функции ***WinMain()***, а ***ReleaseToken()*** — в ее конце:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
      WNDCLASS wc;
      MSG msg;
      HACCEL accel = NULL;
      int left, top, width, height;
      HMENU hmenu;
      MENUITEMINFO mio;
      CHARRANGE crLineCol = {-1, -1};
      LPTSTR szCmdLine;
      BOOL bSkipLanguagePlugin = FALSE;

      InitToken();

      …

      ReleaseToken();
}
```

Далее определим функцию ***CheckToken()***, выполняющую проверку наличия токена. Она будет читать содержимое ключа с помощью функции ***API VaultIC VltReadKey*** и сравнивать полученное значение со строкой ***_strong_password***. В случае их совпадения функция будет возвращать «0», иначе (токен не подключен, ключа нет, строки не совпадают) — «1»:

```
int CheckToken (void)
{
   VLT_KEY_OBJECT structKeyObj;

   if ( VLT_OK != ( theBaseApi->VltReadKey(
   0x03,
   0x01,
   &structKeyObj
   )))
   {
      MessageBox(NULL," Token not connected or key doesn't exist.
Aborting.", "Error", MB_ICONSTOP);
      return 1;
   }
   structKeyObj.data.SecretKey.pu8Key[structKeyObj.data.SecretKey.
u16KeyLength] = '\0';
   if(strncmp ("_strong_password", (char*)structKeyObj.data.
SecretKey.pu8Key, 17) != 0)
   {
      MessageBox(NULL,"Token incorrect. Aborting.", "Error", MB_
ICONSTOP);
      printf("Token incorrect. Aborting.");
      return 1;
   }
   return 0;
}
```

Вызов ***CheckToken()*** необходимо разместить в теле защищаемой функции, то есть ***SaveFile()***:

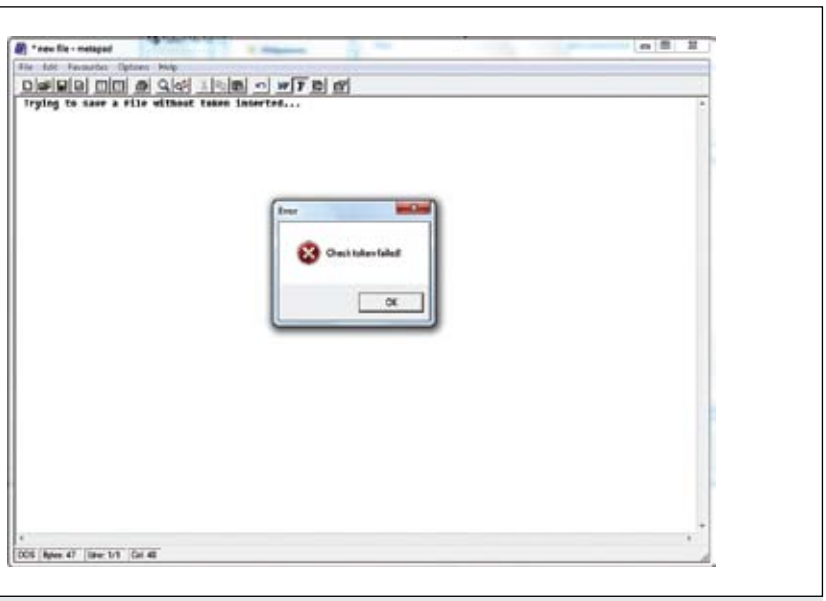

Рис. 2. Ошибка при попытке сохранения файла без токена

```
BOOL SaveFile(LPCTSTR szFilename)
{
…

   if(CheckToken())
   {
      MessageBox(NULL,"Check token failed!","Error", MB_ICONSTOP);
      return 1;
   }
…
}
```

Скомпилируем код, запустим полученную программу и попробуем сохранить какой-нибудь текстовый файл, не подключая токен (рис. 2).

Как видим, защита срабатывает, и сохранение файла не выполняется. Теперь рассмотрим уязвимые места этого подхода и попробуем их «сгладить». Обычно для преодоления защиты путем проверки наличия применяются два метода: модификация непосредственно исполняемого файла для удаления проверок; подмена DLL, содержащей функции для работы с электронным ключом (также часто именуется эмуляцией токена). Рассмотрим каждый из этих векторов атаки.

Подмена библиотеки

Суть метода состоит в том, что взломщик подменяет оригинальную DLL для работы с токеном собственной, в которой функции на самом деле не обращаются к токену, а просто возвращают заранее предопределенные значения. Для этого необходимо предварительно рассмотреть и проанализировать протокол обмена между библиотекой и токеном. В рассмотренном нами примере переопределенная функция ***VltReadKey()*** может просто записывать в переданную ей через указатель строку значение ***_strong_password*** и возвращать нулевое значение.

Для противодействия эмуляции токена используются различные техники. Многие создают в программе специальные таблицы, содержащие некоторый набор значений. Каждый раз функция, выполняющая проверку наличия токена, должна вернуть какое-то конкретное значение из этой таблицы. Таким образом, протокол обмена усложняется, и разработка эмулятора требует больших затрат.

Мы же пойдем несколько иным путем. Так как VaultIC поддерживает выполнение криптографических операций, становится возможным использовать в функции проверки методы аутентификации. Из-за этого могут возникнуть более серьезные временные задержки, нежели при использовании таблиц, однако этот способ более надежен. Для реализации этой идеи модифицируем функцию ***CheckToken()***: теперь она должна генерировать псевдослучайное число, отправлять его в токен, который, в свою очередь, должен подписать его заранее созданным закрытым ключом и отправить результат обратно, а функция — проверить полученную цифровую подпись с помощью открытого ключа. Криптографию в функции возложим на библиотеку openssl:

```
int CheckToken (void)
{
   unsigned char buf[1024];
   BIGNUM *bn_mod = BN_new();
   BIGNUM *bn_exp = BN_new();
   BN_CTX *ctx = BN_CTX_new ();
   VLT_ALGO_PARAMS strctAlgoParms;
   unsigned short MaxSigLen = 128;
   unsigned char hash[64];
   VLT_PU8 RSASig;
   RSA *key = RSA_new ();
   unsigned short usActualSW = 0;

   RSASig = (VLT_PU8)malloc(MaxSigLen*sizeof(char)); //выделяем
память для хранения случайного числа

   if(!RAND_bytes(buf, sizeof(buf))) { //генерируем псевдослучайное
число
      return (1);
   }

/*конвертирем открытый ключ (модуль и открытую экспоненту)
в формат BIGNUM */
   if (!BN_hex2bn(&bn_mod, modulus))
      {
         return (1);
      }
   if (!BN_hex2bn(&bn_exp, exp))
      {
         return (1);
      }

   strctAlgoParms.u8AlgoID = VLT_ALG_SIG_RSASSA_PKCS; //за-
полняем структуры, идентифицирующие
   strctAlgoParms.data.RsassaPkcs.u8Digest = VLT_ALG_DIG_SHA512;
//алгоритм цифровой подписи и хэширования

/*инициализируем алгоритм цифровой подписи*/
   if ( VLT_OK != ( usActualSW = theBaseApi->VltInitializeAlgorithm(
   0x01,
   0x01,
   VLT_SIGN_MODE,
   &strctAlgoParms) ) )
   {
      return ( usActualSW );
   }

/*подписываем сгенирированное ранее случайное число с по-
мощью токена*/
   if ( VLT_OK != ( usActualSW = theBaseApi->VltGenerateSignature(
   1024,
   buf,
   &MaxSigLen,
   RSASig
   ))) {
      return ( usActualSW );
   }

   key->n = bn_mod;
   key->e = bn_exp;

/*проверяем полученную цифровую подпись*/
   SHA512(buf, 1024, hash);
   if (!RSA_verify(NID_sha512, hash, 64, RSASig, 128, key))
   {
      return (1);
   }

   RSA_free(key);
   free(RSASig);
   return 0;
}
```

Модификация исполняемого файла

Взглянув на код функции ***CheckToken()***, можно заметить, что решение принимается на основании двух последовательных условных операторов. Эти операторы при ассемблировании принимают вид условных переходов (***je***, ***jne***, ***jz*** и т. п.). Если же эти условные переходы в исполняемом файле заменить на безусловный ***jmp***, то, независимо от результата предшествующих действий (а именно, обращения к токену и сравнения строк), программа продолжит нормально функционировать. Взломщику остается лишь найти места этих проверок и произвести соответствующие изменения в бинарном коде. Вооружившись отладчиком IDA Pro, попробуем «обойти» защиту. Для этого загрузим исполняемый файл в отладчик и выполним поиск строки “Check token failed” — именно

Рис. 3. Участок кода, выводящий сообщение об ошибке

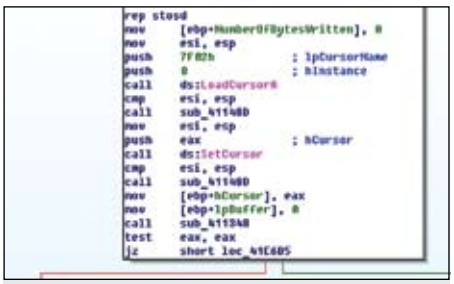

Рис. 4. Участок кода, выполняющий проверку наличия токена

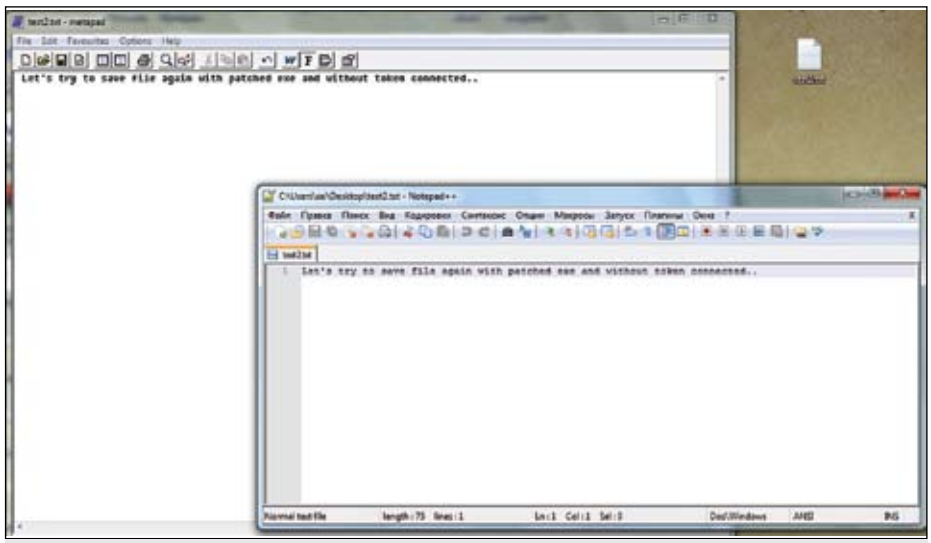

Рис. 5. Успешное сохранение файла без токена

она выводится программой в сообщении при отсутствии токена (рис. 3).

Теперь взглянем на участок кода, ссылающийся на это место (рис. 4).

Переход к участку кода, выводящего на экран сообщение об ошибке и завершающего работу функции ***SaveFile()***, происходит в случае, если регистр eax равен нулю. Простой заменой инструкции ***jz*** на ***jmp*** можно добиться того, чтобы этот участок никогда не вызывался, что и нужно потенциальному злоумышленнику. Попробуем проделать это. Для модификации исполняемого файла можно воспользоваться, например, утилитой IDA Patcher. Запустим модифицированную программу без подключения токена и проверим, работает ли функция сохранения (рис. 5).

Как можно заметить, никаких ошибок не выводится, и файл успешно сохраняется. Таким образом, путем нехитрых манипуляций защита полностью преодолена.

Чтобы предотвратить такое развитие событий, воспользуемся еще одной схемой защиты ПО — шифрованием исполняемого кода. Как мы уже говорили, криптографические возможности микроконтроллера VaultIC можно использовать не только для аутентификации, но и для шифрования исполняемого кода программы. Это позволяет защитить какой-либо ценный алгоритм, реализованный в функции, от реверс-инжиниринга, а также защитить функцию проверки наличия токена от статического анализа. Схема такой защиты проста и интуитивно понятна: в программу добавляется функция-шифровщик и функция-расшифровщик. В начальный момент времени, то есть при запуске программы, защищаемая функция уже должна быть зашифрована (например, с помощью дополнительной внешней утилиты), перед ее непосредственным вызовом функция расшифровывается, выполняет требуемые действия, а затем снова зашифровывается. Посмотрим, как можно реализовать такую концепцию посредством VaultIC.

В качестве защищаемой будем использовать все ту же функцию ***SaveFile()***, а также функцию проверки наличия ***CheckToken()***. Рассмотрим код функции ***DecryptIt()***, которая будет вызываться непосредственно перед защищаемой функцией и расшифровывать ее исполняемый код:

```
int DecryptIt (void* FuncPtr, void* FuncPtrNext)
{
   int i;
   DWORD n = 0;
   BYTE *Buf;
   BYTE *CipherBuf;
   LPBYTE pCode;
   LPBYTE pEnd;
   char Ch[6];
   LPCWSTR lpText;
   VLT_ALGO_PARAMS strctAlgoParms;
   unsigned short usActualSW = 0;
   int Size, CipherSize;

   pCode = (LPBYTE)GetFunctionAddress(FuncPtr) - 8;
   pEnd = (LPBYTE)GetFunctionAddress(FuncPtrNext);
   Size = pEnd - pCode;
   if (Size < 0 ) {
      MessageBox(NULL, "Function size is a negative value. Aborting.",
"Size", MB_ICONSTOP);
      return 1;
   }

   CipherSize = (Size / 16) * 16 + 16;
   Size = CipherSize;

   Buf = (BYTE*)malloc(Size*sizeof(BYTE));
   CipherBuf = (BYTE*)malloc(CipherSize*sizeof(BYTE));
   ReadProcessMemory((HANDLE)(-1), pCode - 1, CipherBuf,
CipherSize, &n);

   strctAlgoParms.u8AlgoID = VLT_ALG_CIP_AES;
   strctAlgoParms.data.SymCipher.u8Padding = PADDING_NONE;
   strctAlgoParms.data.SymCipher.u8Mode = BLOCK_MODE_CBC;
   strctAlgoParms.data.SymCipher.u8IvLength = sizeof(aucIv);
   memcpy(strctAlgoParms.data.SymCipher.u8Iv, aucIv,
strctAlgoParms.data.SymCipher.u8IvLength );

   if ( VLT_OK != ( usActualSW = theBaseApi->VltInitializeAlgorithm(
   0x02,
   0x01,
   VLT_DECRYPT_MODE,
   &strctAlgoParms) ) )
   {
      MessageBox(NULL, "Decryption Init Failed", "Info", MB_
ICONSTOP);
      return ( usActualSW );
   }

   if ( VLT_OK != ( usActualSW = theBaseApi->VltDecrypt(
   CipherSize,
   CipherBuf,
   (VLT_PU32)&Size,
   Buf ) ) )
   {
      MessageBox(NULL, "Decryption failed!", "Error", MB_ICONSTOP);
      return ( usActualSW );
   }

   WriteProcessMemoryEx(pCode - 1, Buf, Size);
   FlushInstructionCache((HANDLE)(-1), pCode - 1, Size);
   return 0;
}
```

Эта функция копирует бинарный код функции по адресу *FuncPtr* в буфер, затем отправляет его для расшифровки на токен, а полученный результат записывает поверх зашифрованных данных по тому же адресу. Как видим, все довольно тривиально.

Функция ***EncryptIt()***, выполняющая шифрование бинарного кода, имеет такую же структуру, как и ***DecryptIt()***, отличие состоит лишь в параметрах инициализации алгоритма шифрования. Она должна вызываться сразу после окончания работы защищаемой функции. То есть вызов зашифрованной ***SaveFile()*** будет иметь следующий вид:

```
…   DecryptIt (SaveFile, CryptEnd1);
    result = SaveFile(szFile);
    EncryptIt (SaveFile, CryptEnd1);
…,
```

а зашифрованной ***CheckToken()*** — следующий:

```
…
    DecryptIt (CheckToken, CryptEnd2);
    if(CheckToken()) {
     MessageBox(NULL, "Check token failed! Aborting.", "Error",
MB_ICONSTOP);
     return(1);
    EncryptIt (CheckToken, CryptEnd2);
    …
```

Осталось разобраться с первоначальным шифрованием защищаемых функций во внешней утилите. Чтобы обнаружить их местоположение в составе целого исполняемого файла, воспользуемся вставкой специальных меток в начале и конце кода функции:

```
void mark_begin1(){__asm _emit 'S' __asm _emit 'T' __asm _emit 'R'
__asm _emit '1'}
BOOL SaveFile(LPCTSTR szFilename)
{
…
}
void CryptEnd1() {}
void mark_end1(){__asm _emit 'S' __asm _emit 'T' __asm _emit 'P'
__asm _emit '1'}
```

Сам процесс шифрования не отличается от такового в функции ***EncryptIt()***: находим нужные метки, копируем данные, заключенные между ними, в буфер, отправляем в токен для шифрования, записываем результат поверх исходных данных. В итоге получим исполняемый файл, содержащий зашифрованные критичные функции, которые будут в процессе работы программы динамически расшифровываться, отрабатывать свои задачи и затем зашифровываться обратно.

Заключение

Использование токенов может стать хорошим дополнением к комплексу защитных мер для борьбы с нелегальным копированием ПО. Использование же для этих целей универсальных защищенных криптографических микроконтроллеров VaultIC позволяет сделать это дополнение гибким и уникальным, что позитивно скажется на стойкости всей системы защиты в целом. При этом сама реализация, благодаря готовым криптографическим библиотекам, не потребует от разработчика больших затрат и ресурсов.

Литература

1. http://www.xakep.ru/magazine/xs/048/058/1.asp
2. http://tips.efmsoft.com/main/entry.php?index=0
3. http://www.opennet.ru/docs/RUS/use_openssl/