Palm OS® Platform: Software Protection

Aaron ARDIRI

Aaron Ardiri, ***ardiri.com*** *,*
Rörstrandsgatan 12, SE 113-40 Stockholm, Sweden
Tel: +46 70 656 1143; Email: aaron@ardiri.com

Abstract. Software piracy is the unauthorized copying of software. It occurs in many forms, however, the main forms are that of illegal copying and unauthorized modification of software to avoid registration systems or notices. When a user purchases software, they enter into a contract with the vendor and become a licensed user of the program. This license does not transfer ownership of the program, but it gives the user the right to use it. This paper will present issues that authors of software on the Palm OS® [1] Platform should be aware of - in relation to software licensing, electronic software distribution websites, anti-cracking techniques, and the personal experiences of a developer who went underground into the piracy community to exploit the techniques used in software piracy.

1. Introduction

Protecting software to fight piracy and unauthorized tampering is a difficult task. Developers must first decide on a licensing scheme that best fits their business model, and then implement it. There are many different schemes and implementations possible. The most common types will be discussed in this paper, as well as discussion about various techniques that can help deter or slow down the time it takes for the application to enter the piracy scene.

Unfortunately, describing a software protection scheme is a double-edged sword. On one side, it is extremely helpful for developers to have as much detailed information as possible in order to create a robust system. On the other, when this same information falls into the hands of those who wish to defeat the protection, it becomes a trivial task. To reduce the ease by which prying eyes can use such information, it is preferable to discuss the theory behind a protection scheme, and not reveal too deeply the code level implementation.

To defeat software copy protection, familiarity with low-level assembly language and understanding of various development tools, including a debugger is a requirement. To fully understand how an application is compromised, it is necessary for developers to also be familiar with these topics.

2. Software Licensing

All computer software is distributed with a particular license, which is imposed when the user obtains a copy of the software. There are a variety of license types, well defined by the Free Software Foundation [3], but in summary they are:

2.1 Free Software

Free Software is software that comes with the permission to use, copy, and distribute, either as-is or with modification for free or with a fee. The most important factor of free software is that the source code must be provided with it. Free software is the matter of freedom, not price. The GNU General Public License (GPL) is an example of a true free software license.

2.2 Freeware

Freeware is software where the developer does not ask for a licensing fee, and the software may be distributed freely in an unmodified state. The source code is generally not available and the product must be used on an as-is basis. In the case where the source is available, the new developer must ask permission from the original developer to modify and redistribute the software.

2.3 Shareware

Shareware is software similar to freeware; however, the developer asks users who continue to use the software after the evaluation period has expired to pay a license fee. The source code is not available, however, the developer encourages distribution of the software as-is. The developer may implement a number of systems that will allow the user to try out the software.

The developer may do various things to encourage users to obtain registration for their products:

- disable application features,
- limiting usage of particular feature,
- "faith" system; offer full version with no concept of registration,
- present reminders of software registration status (dialogs, messages et al), or
- force the user to wait intentional time periods when performing tasks

The technique for doing this differs from developer to developer, and many other techniques exist.

2.4 Commercial

Commercial software is software that has been developed by a business that aims to make money from any use of the software. Commercial software may not be redistributed, and, in many cases, a smaller, crippled demonstration copy of the software is made available for evaluation purposes.

3. Electronic Software Distribution

Palm OS® Platform software can be distributed either in retail or over the Internet using an electronic software distribution (ESD) website that handles credit card transactions directly, take out a small processing fee, and pass revenues to the developer of the applications that are purchased.

3.1 The Purchase Process

ESD websites allow the consumer to navigate through a software catalogue and place items they wish to purchase into a shopping cart. They proceed to a checkout and finalize their order by supplying their email, credit card details, and HotSync™ user name or other pieces of information required by the developer of the applications they purchased. An invoice is generated, and the software developer is sent an email to acknowledge the purchase of their software item.

In many cases, the consumer is not given the product at the time of purchase. Developers are emailed a report of purchases and must then personally contact their consumers and provide the appropriate registration files and or keys. This causes some lengthy delays, especially when dealing with developers overseas, and is not acceptable to the consumer.

3.2 RealTime Fulfillment™

RealTime Fulfillment™ is the ability to provide consumers who purchase a software program from an electronic software distribution site instantaneous access to the registered version directly after purchase. It can be done in two ways: provide a download of a full version, or present an unlocking code that when entered, will enable previously disabled or time limited features of the program.

In Q1 2001, PalmGear H.Q. [5] will provide the ability for developers to generate unlocking codes to users during the execution of the invoice for purchase of software. `jCode`, the system to be provided by PalmGear H.Q., allows developers to edit, compile and test key generation algorithms via a web browser without the installation of a development kit.

The introduction of this system will allow developers to use key based systems for registration, without forcing the purchase-receive delay and manual processing of orders by developers. A demonstration of the system is available at the following website:

http://www.ardiri.com/palm/jCode/

Since the algorithms are supplied to the ESD website, the developer is required to establish trust. However, supplying an algorithm differs no way from supplying a serial number of full version of the product, which a number of developers already do.

4. Registration Techniques

The most important decision to make when developing an application for resale is to decide how the registration system should work with your application. It is important to consider all options that are available and choose the one that will encourage the users to purchase the application. The choice of the wrong technique may not allow the user of the application to satisfactorily complete a trial on your product – and deter them from purchasing it.

4.1 Copy Protection Database Bit

The Palm OS® Platform implements its own form of copy protection which can prevent an application or database from being beamed between devices from within the application launcher. An application is essentially an executable resource database, and the programmer can set certain properties of the application using a simple application programmer interface (API) call [4].

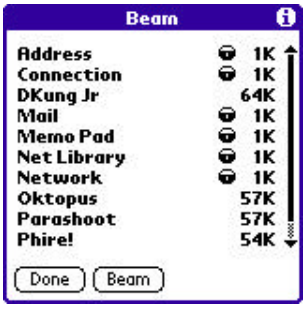
```
Err DmSetDatabaseInfo(UInt16 cardNo, LocalID dbID, const Char* nameP,
                      UInt16* attributesP, UInt16* versionP,
                      UInt32* crDateP, UInt32* modDateP,
                      UInt32* bckUpDateP, UInt32* modNumP,
                      LocalID* appInfoIDP, LocalID* sortInfoIDP,
                      UInt32* typeP, UInt32* creatorP);
```

Every database has particular attributes, and can be set using the fourth (4th) parameter of this system call. Using the following code, it is possible to add this type of copy protection to your application. Newer versions of the development compilers support setting this bit at compile time.

```
{
  UInt16  card, attributes;
  LocalID dbID;

  SysCurrAppDatabase(&card, &dbID);
  if (dbID != NULL) {
    DmDatabaseInfo(card, dbID, NULL, &attributes, NULL, NULL, NULL,
                   NULL, NULL, NULL, NULL, NULL, NULL);
    attributes |= dmHdrAttrCopyPrevention;
    DmSetDatabaseInfo(card, dbID, NULL, &attributes, NULL, NULL, NULL,
                      NULL, NULL, NULL, NULL, NULL, NULL);
  }
}
```

The application launcher checks to see if this bit is set, and will prevent any beaming of the database. A protected application is marked using a lock icon in the beam dialog (as seen on right).



The idea is simplistic and provides limited protection, however, many third party file management utilities do not check this attribute – and beaming of databases (including applications) occurs very easily. It is also very easy to write a small Palm application that resets this bit on all databases within the device, hence fooling the mechanism.

Using this technique may actually be more damaging to the developer. Distribution of a product is a very important issue – the ability for the user to simply beam the application to another provides free "word of mouth" advertising of the product. Registered versions can be beamed to other users and made demonstration versions if the application is designed correctly.

4.2 Serial Number

A serial number is a number or string that is common between all registered users of a particular piece of software. When the user enters this serial number, the program has all its features activated. The developer will normally provide a dialog for the user to supply a serial number or string, and the handling of that entry is performed in a manner similar to the following:

program.c:

```
#define SERIALNUMBER         12345
#define incorrectSerialAlert 1001

{
  codeEntered = StrAToI(strCode);
  if (codeEntered != SERIALNUMBER)
    FrmAlert(incorrectSerialAlert);
  else
    registered = true;
}
```

This code will produce assembler similar to the following:

program.s:

```
                ...
2F03            MOVE.L  D3,-(A7)
4E4FA0CE        TRAP    #15
                DC.W    sysTrapStrAToI
4FEF000C        LEA     12(A7), A7
0C403039        CMPI.W  #12345!$3039,D0
6708            BEQ     L9
3F3C03E8        MOVE.W  #1000!$03E8,-(A7)
4E4FA192        TRAP    #15
                DC.W    sysTrapFrmAlert
7000      L9    MOVEQ   #0,D0
                ...
```

The serial number is right in front of you! Hidden in that mess of assembly code is a comparison of the `D0` register (which contains the result from the `StrAToI` call) with a constant value. In this case, the value being compared is `0x3039`, or `12345`. This is the case when dealing with simple numbers, if comparing strings the `StrCompare` API [4] call is used, and the string being compared has its address loaded using an `LEA` instruction (to a label in the code) prior to being called. Searching the code for that label will lead you straight to memory location containing the required string.

The developer must place a lot of trust on their users to ensure that the serial number is not broadcast to their friends or the world. A 'good faith' user may purchase the product and simply pass on the information onto others. Pooling for the cost of an application may occur in this case.

4.3 Nag Screen

A "Nag Screen" is a dialog that appears when you start an application reminding you to register. The purpose is to constantly nag the user to buy the software until they get to a point where they are sick and tired of seeing it.

Implementation of a nag screen is simple, using the following API [4]:

```
UInt16 FrmAlert(UInt16 alertID);
```

These types of applications are very vulnerable to attack and modification, as doing the adjustment is a very simple task for even the beginner cracker. Lets consider the following code snippet:

program.c:

```
#define nagScreenAlert 1000

{
  FrmAlert(nagScreenAlert);
}
```

This code will produce assembler similar to the following:

program.s:

```
                ...
3F3C03E8        MOVE.W  #1000!$03E8,-(A7)
4E4FA192        TRAP    #15
                DC.W    sysTrapFrmAlert
                ...
```

The simple task of replacing the call to the API is all that is required to prevent the alert from appearing on the screen. The crack would be as simple as changing the `TRAP` call (`0x4E4FA192`) into a series of `NOP` instructions (`0x4E714E71`) – two `NOP` statements are required as the `TRAP` uses four bytes and a `NOP` requires only two bytes. The resulting code would look as follows:

program.s:

```
                ...
3F3C03E8        MOVE.W  #1000!$03E8,-(A7)
4E71            NOP
4E71            NOP
                ...
```

The nag screen will never appear again once these modifications are in place, which can be done using your favourite HEX (binary) editor.

4.4 Code Generation Systems

The most common mechanism for registration is the use of an application based code generation algorithm that allows the developer to assign each user a pseudo-unique code that allows them to gain access to all features of the application.

Since each user has a unique code specifically for them, transfer of the application can occur without posing a threat to the impact of sales. In many cases, the "beam" factor will allow the application to spread further and increase sales.

The Palm OS® Platform provides an inbuilt pseudo-unique piece of information. Each user must HotSync™ their device in order to perform backups, update the data on their device, or install software. This information is pseudo-unique as in many cases two people in close proximity of each other would not use the same HotSync™ user name (if they did, they wouldn't perform a HotSync™ on the same machine). It is possible to obtain the HotSync™ username within an

application using the following code [4].

```
#include <System/DLServer.h>

{
  CharPtr username =
    (Char *)MemPtrNew(dlkUserNameBufSize * sizeof(Char));
  DlkGetSyncInfo(NULL,NULL,NULL,&username,NULL,NULL);

  ...

  MemPtrFree(username);
}
```

The HotSync™ username causes many small problems. Most users don't even know what it is, yet alone are capable of typing it in correctly when they purchase your product. Care needs to be taken in respect to punctuation, capitalization, spacing and localization of the HotSync™ user name. The most flexible manner to deal with this problem is to translate the binary representation into another form that the user can supply, as can be done with development kits like RegCode [6].

The Palm III series of devices (running Palm OS® 3.0) introduced another form of unique identification, the flash ROM serial number. In the case where the device contained a flash ROM chip, it was possible to obtain a unique serial number for each user, using the following code [4]:

```
{
  CharPtr serialNo;
  UInt16  serialLength, returnVal;

  returnVal =
    SysGetROMToken(0,sysROMTokenSnum,&serialNo,&serialLength);
  if ((!retval) && (serialNo) && ((UInt8)*serialNo != 0xff)) {
    ...
  }
}
```

However not all Palm OS® Platform devices (Handspring [2], IIIe, m100) contain flash ROM memory chips. They contain what is known as mask ROM chips, which do not have this serial number and are not user upgradeable. It also poses a problem when the user damages or loses their device – a replacement would have a different flash ROM serial number and require the user to re-register or re-purchase the application. Palm, Inc. has not committed to supplying this information on its own or licensee devices.

As an alternative, the developer may allow the user to manually enter or provide a string that is used to generate the unlocking key value. Since it is not tied specifically to a single device, it allows a user to use the same username/key combination on a number of devices (including their friends'), but also allows users to break devices (new ROM chip or HotSync™ username) and stay registered without having to go through the process of getting a new registration (they did pay remember).

Once the user information is obtained, the developer writes a function similar to the following:

```
UInt16 generateRegistrationCode(CharPtr username)
{
  UInt16 code, i;

  code = 0xCAFE;
  for (i=0; (i < dlkUserNameBufSize) && (username[i] != '\0'); i++) {
    code += (((username[i] & 0xAA) << 8) | (username[i] & 0x55));
    code  = ((code << 1) | ((code & 0x8000) >> 15));
  }

  return code;
}
```

It is important to make the code generation as non-simplistic as possible. As soon as someone figures out how the code is generated – a key generator will be available. One of the most tedious tasks for someone who wants a free copy of your software is to sit down and figure out how this algorithm works. The more cryptic it is, the longer it will take them to figure out.

When a key generator is available, in many cases subsequent releases of an application do not need to go through the cracking phase again – as they will then have a tool to bypass all protection until the key generation algorithm changes.

Another approach is to try the reverse – supply the user with a code that is then used to generate a string that can to be compared against the string that is gathered on the device. This technique makes it slightly harder to generate a key generator, as the algorithm to convert a string to a number is different from converting a number to a string – which means the reuse of your code generation algorithm won't occur.

4.5 Usage Counter

"Trial Periods" are very common in applications, and may be limited to a number of application executions or a pre-defined period of time.

A common usage counter example is the "30 day trial period".

This type of registration allows a user to gain full access to the application during the limited trial period, and can be implemented as follows:

```
typedef struct
{
  UInt32    keyData;    // the time the application was first started
  UInt8     keyValue;   // the number of days remaining in the trial
} KeyType;

{
  KeyType *key = NULL;

  // load the key from somewhere
  ...

  // a key exists, lets do our checking
  if (key != NULL) {
    UInt32 diff = TimGetSeconds() - key->keyData;

    // 30 days is over?
    if (diff >= 0x00278D00) key->keyValue = 0; // 0x00278D00 = 30 days
    else
      key->keyValue = 30 - (diff / 0x00015180);// 0x00015180 =  1 day
  }

  // no key, create it.
  else {
    key = (KeyType *)MemPtrNew(sizeof(KeyType));
    key->keyData  = TimGetSeconds();
    key->keyValue = 30;
  }

  // save the key somewhere
  ...

  // clean up
  MemPtrFree(key);
}
```

The application must store some information somewhere, such that the remaining time available can be determined. The "`KeyType`" structure contains all the relevant information required for an application to manage its usage based on time. It keeps track of the first time the application was executed, and the number of days remaining in the trial.

Where does one store this key information?

The answer to this question is at the discretion of the developer. It could be stored in one of the following places:

- saved preference
- registration database
- registration manager software

The Palm OS® Platform has a very efficient application manager such that when an application is deleted, all the associated preferences and databases are deleted as well.

If this key is stored in a database or preference that shares the same creator ID as the application, it will be deleted when the user removes the application from the device. Re-installation of the software will allow them to regain demonstration access.

Using a creator ID other than the application is known as "shadowing" behind another application. The problem with this technique is that even if the user has no intension of reinstalling the application – a small amount of data will remain on the device. Leaving data behind in this manner is seen negatively, as memory on the devices themselves is already limited.

Is forcing the user to remove the application and re-install it enough of a hassle to make them buy?

Another technique would be to introduce a software registration manager into the device. The developer could use this storage space to keep necessary information like this, but it too, will also be subject to deletion and takes up valuable memory space.

5. Anti-Cracking Techniques

The art of being successful with anti-cracking is the ability to delay or make the process of generating a pirate version or modification of your application. A number of techniques exist for doing this, however they are most powerful when used in conjunction with each other – making a "can of worms" for prying eyes to deal with.

5.1 Special Builds

A common approach used by a few developers is to build two or more separate versions of the product for distribution. A smaller, feature limited version for demonstration purposes and a larger full feature version for those registered users.

Although regarded as one of the best mechanisms to protect your software, as its not possible to modify the demonstration version to be registered, it does not allow for beaming of the application or prevent distribution between people who have the registered product. Just as the demonstration version is made available for download on the Internet, so will the registered version.

5.2 Registration Check

If an application has implemented a registration system where some lockout is performed, the application must check if the user has registered or if the demonstration period has completed. The logic is simple; if they have registered let them continue – if not, show a dialog informing the user that their trial is over and they must purchase the product.

Many application developers opt for the simple solution, by writing a single function that, based on what the application knows, returns a `true` or `false` value that represents the registration status.

```
program.c:
Boolean checkRegistered()
{
  Boolean result = false;

  // check user registration (using appropriate technique)
  ...

  return result;
}
```

This code will produce assembler similar to the following:

```
program.s:
4E560000      LINK    A6,#0
              ...
57C0          SEQ     D0
4400          NEG     D0
4E5E          UNLK    A6
4E75          RTS
```

The return type to a function on the Palm OS® Platform is stored in the `D0` register. After the application calls the function, it compares the result stored in this register to determine what needs to be done next.

```
program.s:
4E560000      LINK    A6,#0
              ...
7001          MOVEQ   #1,D0
4E5E          UNLK    A6
4E75          RTS
```

Replacing the assembler instruction before the `UNLK` command with a `MOVEQ #1,D0` instruction (`0x7001`) makes this routine always return a `true` result. This means, whenever a check is performed, the application thinks it is always registered – and the program continues on as appropriate.

The modular approach to handling this makes it a simple target, as the check is done in one place.

Its pretty easy to prevent this from happening, using the `inline` attribute, which tells the compiler to not make the function call a subroutine and instead place the contents on the function where it is being called.

```
program.c:
inline Boolean checkRegistered()
{
   ...
}
```

Each call to the `checkRegistered()` function will result in the same code being duplicated in many places through out the application – which means more patches are required.

Another technique is to not use a `Boolean` result variable. By returning an integer value, it is not known what the real return result should be – making it a guessing game, to some extent (it will always be able to trace the call somewhere – it just takes longer).

5.3 Palm OS® Emulator (`POSE`) Detection

The Palm OS® Emulator is a very important tool as it provides a safety net for reckless tampering of software without the risk of destroying data or causing damage to the device. When used in

conjunction with a powerful debugger tool, it proves an excellent software debugging and patching environment. Without these tools, it would be very tedious and risky to perform software patching. You can detect if your application is running on POSE using the following two techniques:

```
Boolean onPOSE()
{
  UInt32 value;
  Err    err;

  // works on all versions of the Palm OS
  return (FtrGet('pose', 0, &value) == errNone);
}
```

or:

```
#include "HostControl.h"                    // distributed with POSE

Boolean onPOSE()
{
  // works only on versions of the Palm OS >= 3.0
  return (HostGetHostID() == hostIDPalmOSEmulator);
}
```

With careful design, it may be possible to implement some fancy craftwork, making it impossible or very difficult for someone to tamper with the software on the emulator. Create a few false leads for the people with prying eyes if the emulator is detected. It's great knowing where the needle is and watch them search in the wrong haystack.

Palm Debugger may also be connected to a real device for debugging purposes, eliminating the need for the Palm OS® Emulator.

5.4 Code Checksum

The ultimate defence against software patching is to be able to detect the presence of modification. A checksum, though normally used to verify large chunks of data that are transmitted between sources, can also be used to determine if a modification has been made to an application. The API [4] provides a routine, at system level to perform a CRC16 level checksum.

```
{
  MemHandle  codeH;
  void      *codeP;
  UInt32     codeSize;
  UInt16     checkSum;

  // obtain a reference to the code0001.bin resource
  codeH    = DmGet1Resource('code', 0x0001);
  codeSize = MemHandleSize(codeH)
  codeP    = (void *)MemHandleLock(codeH);

  // determine the checksum of the code segment
  checkSum = Crc16CalcBlock(codeP,codeSize,0);

  MemHandleUnlock(codeH);
}
```

The value returned from the system call can then be used to determine the validity of the code resource that was checked. If the resulting checksum is not what is expected, the application can bring up a dialog warning the user or force the application to be unregistered.

It may, however, be more appropriate for the developers to write their own checksum routine. The `Crc16CalcBlock` routine uses a well-known checksum algorithm, and using a standard routine makes it much easier to calculate a new checksum result value if a patch is applied to the

application. In the case where a custom routine is written, it must also be reverse engineered to determine the correct checksum value.

5.5 Patch Detection

Another form of verifying the binary of the application is to analyse it in search of various known assembler op-codes that have been inserted into the distributed application. The two most powerful assembler op-codes used in the patching of software are `NOP` (`0x4E71`) and `TRAP #8` (`0x4E48`). In many cases, these op-codes are not found in the release versions of the applications – as they are used mainly to aid in the development of the software.

The following code scans all code segments and determines the existence of these two op-codes:

```
{
  MemHandle  codeH;
  void      *codeP;
  UInt32     codeSize;
  UInt8      patchStatus, i;

  patchStatus = 0;
  for (i=0; i<CODE_RES_COUNT; i++) {

    // obtain a reference to the resource
    codeH    = DmGet1Resource('code', i);
    codeSize = (MemHandleSize(codeH) >> 1);  // we are counting words
    codeP    = (void *)MemHandleLock(codeH);

    // search for a patch (0x4e48 or 0x4e71)
    asm("movem.l %%d0-%%d1/%%a0-%%a1, -(%%sp)" : : );

    asm("move.l  %0, %%a0" : : "g" (codeP));
    asm("move.l  %0, %%a1" : : "g" (&patchStatus));
    asm("move.l  %0, %%d0" : : "g" (codeSize-1));

    asm("
         move.w  (%%a0), %%d1
         eori.w  #0xffff, %%d1
         cmpi.w  #0xb18e, %%d1  | check if opcode is 0x4e71
         beq     L01
         cmpi.w  #0xb1b7, %%d1  | check if opcode is 0x4e48
         beq     L02
         bra     L03
  L01:
         ori.b   #1, (%%a1)     | bit one set if NOP
         bra     L03
  L02:
         ori.b   #2, (%%a1)     | bit two set of TRAP #8
  L03:
         addq.l  #2, %%a0
         dbra    %%d0, L01
       " : : );
    asm("movem.l (%%sp)+, %%d0-%%d1/%%a0-%%a1" : : );

    MemHandleUnlock(codeH);
  }
}
```

The above code is specifically designed for use with the PRC-Tools (GNU gcc) development environment; however, it should not be too cumbersome a task to write it to be compatible within the CodeWarrior® development environment.

A reference to each code segment is obtained, and then a linear search is performed looking for the two op-codes (`0x4E71` and `0x4E48`). `XOR'ing` with `0xFFFF` is required to prevent the patch

detection code from finding the two op-codes within it. The scanning code was written in assembler; to force this checking style (required) – good optimizers would not perform it this way.

It is also possible for the developer to use this mechanism to count a specific number of op-codes within the application. Just performing the "detection" and storing it in a `Boolean` result will make it simple prey – just set the value to `false`, and the application won't know any better.

A more advanced technique would be to manually insert a pre-defined number of op-codes into the application, and use that count as a dependency or offset within your application. With the introduction of another op-code, the number changes, and can be used to prevent the program from actually running. Consider the following code:

```
void theCodeToExecuteIfNOTPatched()
{
   ...
}

void theCodeToExecuteIfPatched()
{
   ...
}

{
   void *functionList[] = {
                            &theCodeToExecuteIfPatched,
                            &theCodeToExecuteIfPatched,
                            &theCodeToExecuteIfNOTPatched,
                            &theCodeToExecuteIfPatched
                          };
   void (*function)(void);

   // determine which function to execute
   function = (void *)functionList[opcodeCount % 4];
   function();
}
```

In this example, the code will only execute correctly if the op-count modulus 4 is 3. The introduction of an additional op-code causes the count to change, and call a different function.

5.6 Encrypted Code

An applications algorithm is always open – to the point of being able to read it in the operating systems assembly language. The use of de-compilation utilities makes source code available to prying eyes for inspection – and with enough understanding of the platform specific assembly language these users can determine or modify the logical processing of an application.

One technique to prevent the basic "de-compilation" of your application is to use encryption.

Unfortunately, support for doing this type of operation is not normally built into the compiler. In many cases, a third party utility will need to be written to perform the encryption of code after the normal build process is complete.

```
{
  MemHandle  codeH;
  UInt32     codeSize;
  UInt8      *codeP;

  // duplicate the encrypted resource into memory
  codeH    = DmGet1Resource('data', 0x1111);
  codeSize = MemHandleSize(codeH)
  codeP    = (UInt8 *)MemPtrNew(codeSize);
  MemMove(codeP, MemHandleLock(codeH), codeSize);
```

```
    MemHandleUnlock(codeH);

    // dynamically modify the data stored in the codeP pointer
    // ...

    // execute the function
    {
      void  (*myCode)(void);             // the function specification

      myCode = (void *)codeP;
      myCode();
    }

    // clean up
    MemPtrFree(codeP);
}
```

In the above example, the assumption has been made that within a specific resource there is an encrypted version of a function that needs to be executed. The first step is to obtain a copy of this resource, placing it on the dynamic heap by allocating memory and copying the resource contents. Once a copy of the resource has been obtained, it is possible to perform decryption on the memory. Given this pointer, and once the code it contains has been safely converted into the native platform binary language, a function pointer can be defined and the code can be executed.

What this offers the developer is the assurance that **simple** de-compilation of an application will not reveal the algorithms used within the application. However, with the use of a debugger, a memory dump of the application can be obtained while the program is running. This memory dump can then be de-compiled and viewed by an external party. It does not matter how strong your encryption techniques are – prying eyes can just wait until the decryption is done and then perform this memory dump.

Providing some dependency on a user unique resource places the requirement that at least one purchase must be made of your product to get a copy of the application binary. In many cases, users dealing with illegal copies of software are not willing to pay for even one copy of the software – it may provide more delay in the piracy process.

5.7 Self-Modifying Code

Dynamically changing runtime code is a very interesting and challenging area, however, in many cases it is very difficult to implement and requires a platform specific understanding of how the application will be executed on the platform that is being targeted.

```
{
  // turn off memory semaphore protection
  MemSemaphoreReserve(true);

  // modify memory inside the application (or anywhere else on device)
  // ...

  // turn on memory semaphore protection
  MemSemaphoreRelease(true);
}
```

The use of this technique, when done correctly, can surely throw prying eyes off course and waste a lot of their time (that's a good thing) – however, doing this type of operation can be very dangerous or cause your application to have problems when being used by users. For example, the application will only execute when it is stored in `RAM` – moving it to an external memory card or into flash `ROM` will cause the memory writing operations to fail.

5.8 Code Splitting

An interesting argument in the cracking arena is "*if the registered code isn't there – don't bother*."

It is practically impossible for someone to tamper with your application to make it registered if the code that does actions only available to registered users is not stored in the application itself. Implementing this technique is done in a similar manner to which the encryption of code segments is performed. A reference to a memory location is obtained and then it is executed.

```
{
  DmOpenRef dbRef;

  // try and open the database
  dbRef =
     DmOpenDatabaseByTypeCreator('_key',appCreator,dmModeReadOnly);
  if (dbRef != NULL) {

    MemHandle recordH;
    // get a reference to the first record
    recordH = DmGetRecord(dbRef, 0);

    // execute the function
    {
      void  (*myCode)(void);             // the function specification

      myCode = (void *)MemHandleLock(recordH);
      myCode();
      MemHandleUnlock(recordH);
    }
  }
}
```

In this example, the application tries to locate an external database of a known creator id and database type. If it is found, the first record is obtained, locked and then executed. If the database was not available, the function call would not be made, and the developer could bring up a dialog saying that the functionality is not available.

The demonstration version is the registered version. From a user perspective, they install the application for demonstration purposes and use the software as appropriate. Upon purchase of the application, the user will be provided with an additional file to install onto their device. The nature of the application is such that having the file means you are registered, and not registered if it cannot be found. Users will be able to beam the application to other users, even if it is registered, as the additional database will not be beamed. With additional checks against the HotSync™ username, it is possible to make the database pseudo-unique to each user.

6. CASE STUDY: Liberty - The GameBoy™ Emulator

Liberty [7] is the **first** GameBoy™ Emulator for the Palm OS® Platform. Developed jointly by Michael Ethetton and Aaron Ardiri, Liberty has received outstanding press coverage for achieving what many people had thought, "*could not be done*".

Since Liberty is a commercial product, the descriptions about exactly how the registration scheme operates internally is something for the imagination of the reader. The purpose of describing it is to show the effort required to implement a sophisticated registration system.

6.1 Design

The initial demonstration version of Liberty allowed the user to emulate GameBoy™ game images, which were only `32Kb` in size that mainly consisted of demo or freeware applications. At the time, over 50 GameBoy™ game images were known to meet this requirement.

After receiving a lot of negative user feedback about the limit imposed in the first version, subsequent versions of Liberty provided a trial period of 30-time execution of the application on any sized GameBoy™ game image. Once the trial usage period had expired, the user was restricted to playing GameBoy™ rom images of `32Kb` in size.

6.2 Implementation

To understand the process of how the Liberty application implements its registration system, it is important to understand how the application is structured. Liberty is a multi-segmented application due to its size, but it also makes the act of implementing a registration system easier. A total of seven (7) code segments are defined within the liberty application.

`code0000.bin`	-	system code segment
`code0001.bin`	-	the main code segment
`code0002.bin`	-	registration routines
`code0003.bin`	-	the abstract device layer
`code0004.bin`	-	GameBoy™ emulation routines
`code0005.bin`	-	help system
`code0006.bin`	-	registration system loader
`code0007.bin`	-	encryption key

The registration logic is as follows (in basic terms):

```
LibertyRegistration()
{
  // tampering check
  perform a code scan, looking for NOP, TRAP #8
  if modified
     destroy code0006.bin code segment (self-terminate)
  endif

  // registration loader
  load the code0006.bin resource into the dynamic heap
  using code0002.bin as a key, decrypt it
  execute the decrypted code
  discard the memory used.

  // continue with the program
}
```

To prevent new comers to the piracy scene, there is nothing like having an application crash hard when someone tries to patch the binary using a `NOP` or `TRAP #8` (debugging trap) instruction. If a modification is detected, the application self-destructs. It's annoying and is disabled fairly quickly.

The `code0006.bin` code segment is encrypted based on the `code0002.bin` code segment. It contains the mechanism for loading the registration system and is encrypted to prevent prying eyes from seeing what exactly is going on. What this provides is added protection in the case that the first patch detection systems are bypassed.

```
registrationLoader()
{
  // sanity
  assume no demo, no registration

  // key resource search
  locate the registration key database
  if database found
    load the first record info the dynamic heap
    using code0007.bin and the HotSync™ username, decrypt it
    if demo database
       perform a 30 trial limit check
    end if
    store registered routine for later use
  end if
}
```

Modification to the `code0002.bin` resource causes the registration loader to decrypt incorrectly. The registration loading routine is responsible for locating the registration key, determining if it is a demonstration version or registered, and prepares the system for use. Its purpose is to provide the application with the additional code that is not available in the demonstration version of the application.

When a user purchases Liberty or wants to perform a trial of the application, they must install an additional database, which contains code that gives them full access to the features of the application. The demonstration version and registered version of the product is the same file, and, as a result allows for the beaming of the application amongst users without the need to worry about piracy.

The demonstration key is identical in nature to a full version; however, it has been encoded with a special HotSync™ username that had a 0.01% chance of being used by a normal user.

Illegal distribution between users does not occur as the key file unique to each user. This is accomplished by using the first name of the HotSync™ username to determine a starting offset for the encryption algorithm that uses the `code0007.bin` code segment as its key. The encryption algorithm uses an `XOR` instruction and dynamically modifies the key after each byte is processed.

It all comes down to the following code segment in Liberty:

```
{
  // lock down the first 32K
  globals->emu.ptrPages[0] =
    (UInt8 *)MemHandleLock(DmGetResource(datType,0));
  globals->emu.ptrPages[1] =
    (UInt8 *)MemHandleLock(DmGetResource(datType,1));

  // load and lock the "remainding" rom chunks :))
  {
    GameAdjustmentType adjustType;

    // define the "adjustment"
    adjustType.adjustMode = gameLoadROM;
    adjustType.data.loadROM.pageCount = globals->emu.pageCount;
    adjustType.data.loadROM.ptrPages  = globals->emu.ptrPages;

    // do it! :))
    RegisterAdjustGame(prefs, &adjustType);
  }
}
```

The first `32Kb` of the GameBoy™ game image are locked down regardless of the situation. In the event where the game image is larger than `32Kb`, calling the code stored in the registration key database locks the additional resources. Liberty ensures a game image larger than `32Kb` is not emulated if the application is not registered.

Additional development tools were required in order to perform the post compilation process of encrypting code resources and separating the registered code into an external database file.

The design and implementation of this system required approximately one month of effort.

7. Conclusions

This paper has described a number of issues that a developer must consider when writing applications for distribution on the Palm OS® Platform. The choice of registration technique is always a hard decision, yet it has to be made at some point. A number of issues were discussed here and hopefully provide some help in making this type of decision.

Investigating methods of registration from a technical point of view is an interesting task. It is always possible to make an application take a bit longer to enter the piracy scene.

The ultimate goal is to satisfy the following equation:

```
Time(to perform crack) >= Time(cracker willing to put into it) AND
Threshold(annoyance provided to user) < Threshold(accepted by user)
```

Unfortunately, it is not that simple.

Piracy relies solely on demand – supply is never the issue. The more popular the application, the more likely it will be provided in an illegal form. It is very common for applications such as games and essential tools to find their way to the piracy scene very quickly.

Fighting piracy is a very difficult task.

Every war has a loser. The losers of the piracy war are not the developers or the piracy community – it's the most important person of all, the user. Registration systems were created to force payment from users for the effort that has been performed to produce a particular product. The decision that needs to be made is deciding at what point will the user purchase the application.

It is also good to consider why the piracy scene exists in the first place.

The piracy scene is generally run purely on **pride**. The whole idea of piracy is to get applications for free – I don't see anyone sending pay cheques out to these guys, and some of them put a lot of effort into it. It's like kicking the all-important last second goal in your high school football game.

Why are we doing all this? The answer is very simple – we shouldn't.

I performed the anti-cracking research to satisfy my own personal desires. I wanted to know how it was done – purely out of interest. In the process I put a lot of my own time and effort into it, expanding on my knowledge about the subject. However, at the same time, I provided the fuel needed to keep the piracy scene in existence. The challenge keeps the system ticking. The Liberty registration system, as was described earlier is not trivial, and has been compromised.

Implement what is required to keep **honest users honest** – they are important, don't fight the battle.

References

[1] Palm Computing, http://www.palm.com/
[2] Handspring, http://www.handspring.com/
[3] Free Software Foundation. http://www.gnu.org/philosophy/categories.html.
Categories of Free and Non-Free Software
[4] Palm Computing. http://www.palmos.com/
PalmOS™ SDK 3.5 Reference
[5] PalmGear H.Q, http://www.palmgear.com/
[6] Palm Creations. http://www.palmgear.com/software/showsoftware.cfm?prodID=2997
RegCode Development Kit
[7] Gambit Studios, LLC. http://www.gambitstudios.com/