Cryptographic Applications in the 21st Century

HITBSecConf2007 - Dubai
2nd - 5th April 2007 - Sheraton Creek Hotel
Deep Knowledge Security Conference

Hands on Technical Training
Dual Track Security Conference
Capture The Flag 'Live Hacking' Competition

The Premise

Until a few decades ago data encryption and decryption was handled entirely on a hardware level.

At the time, cryptographic software implementations were simply not feasible, especially when considering the operations involved in asymmetric cryptography.

Breakthroughs in hardware technology over the 80’s and 90’s have resulted in the availability of cheap and powerful personal computers.

This has lead to an array of commercial, free and open source cryptographic software being made available.

The Pitfalls

A dedicated hardware device will perform encryption and decryption operations faster than a general-purpose computer, which has to deal with overheads such as the OS.

When commercial / closed-source cryptographic applications are put to use one can never be certain that:

A) The algorithm being used is a strong one
B) The algorithm has been properly implemented
C) Data encryption and decryption is being performed correctly.

There’s always the dilemma that the software may be modified by a malicious party unbeknownst to its user(s). The result of which could render the entire encryption/decryption process redundant.

Data encrypted with commonly used and popular encryption software is often easy to recognize.

The Pro’s

The availability of tried-and-tested libraries such as OpenSSL, Botan and Crypto++ allow software developers to seamlessly incorporate strong crypto into their applications without having to worry too much about implementation.

It has brought privacy to the man on the street, where before it was generally reserved for wealthy corporations and governments.

It has opened up new avenues and uses for cryptography, for example it’s frequent usage these days as a software registration and protection mechanism.

It’s eliminating dated clear text protocols that are becoming more and more of a security risk.

The Requisites

• Randomization

Decent hardware based random devices provide a source of truly un-guessable random bits.

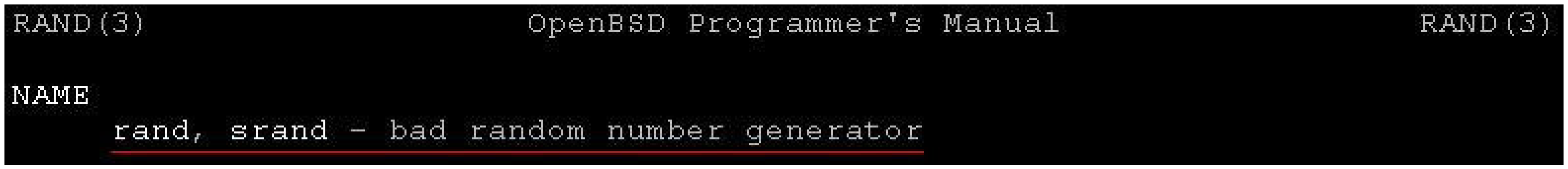
UNIX and Linux variants normally include entropy pools that make use of various system interrupts to acquire random data. These provide random bits strong enough for usage in cryptographic applications.

- What if someone were able to compromise the system and modify the various commonly used random devices (random, urandom, srandom & arandom) to provide predictable bit sequences?

The Requisites

- Randomization

Pseudo-random sources such as the well-known rand() and random() functions should never be made use of as they have no where near the strength required for any cryptographic purpose.



```
RAND(3)                 OpenBSD Programmer's Manual                RAND(3)

NAME
     rand, srand - bad random number generator
```

Instead a strong pseudo-random number generator such as arc4random() (built on the arc4 cipher) provides a decent means of compromise in both speed and levels of randomization. (The arc4random() function is seeded by the arandom device)

The Requisites

• Key generation

Symmetric ciphers require a common key be known amongst all parties involved in the communication process.

- Presents various problems regarding key distribution

- As more people are included in the communication the higher the likelihood becomes of the key being compromised

Asymmetric ciphers overcome these issues.

- Keys could still be compromised if the malicious party were able to view the generation process, or, if there was no generation process to speak of....

The Requisites

• Key generation – RSA 512 example

Generating keys is often a very resource intensive process.

If you consider the example operations required to generate a simple 512bit RSA public/private key:

P (random prime):
75045186078249185930446974332193424379178205846587224462566286795839939672859

Q (random prime):
91000115199631870600433774731076789534402234322822184903285677710662434112979

The Requisites

- Key generation – RSA 512 example

N (P*Q):
6829120578298485794685572331263680971271005860495537613708721773743296210347078919967487609026908791097411553399521780596305888323549458777341223105936961

φ (P-1)*(Q-1):
6829120578298485794685572331263680971271005860495537613708721773743296210346912874666209727970377910348348283185608200156136478914183606812834720732151124

E (Less than N, relatively prime to φ):
65537

The Requisites

• Key generation – RSA 512 example

D (inverse of e modulo φ):
272656314832473532971821544501343479705017225911662476464138
446880032788562090608783612377347258660607252210473220660337
3445032687924809738996800321611705

To encrypt data with the generated public key, the computer would have to calculate:

message to the power 65537 **mod**
682912057829848579468557233126368097127100586049553761370872
177374329621034707891996748760902690879109741155339952178059
6305888323549458777341223105936961

Or pt^E%N

The Requisites

• Key generation

It's clear that a lot of strenuous calculations are required of standard workstations to generate safe keys.

It should also be noted that these days 1024 – 2048 bit keys are the accepted standard.

Is it unthinkable that software vendors might opt for hardcoding primes to be used for public key/key exchange into their applications (perhaps using weak xor encryption/decryption to obfuscate them), simply to save cpu cycles on a customers computer?

Would it be equally unthinkable for vendors hardcode symmetric keys into an application?

The Requisites

• Key generation

Example:

Winace’s CCrypt made use of 2048bit RC6 implementation which proved entirely redundant as key generation was handled abysmally.

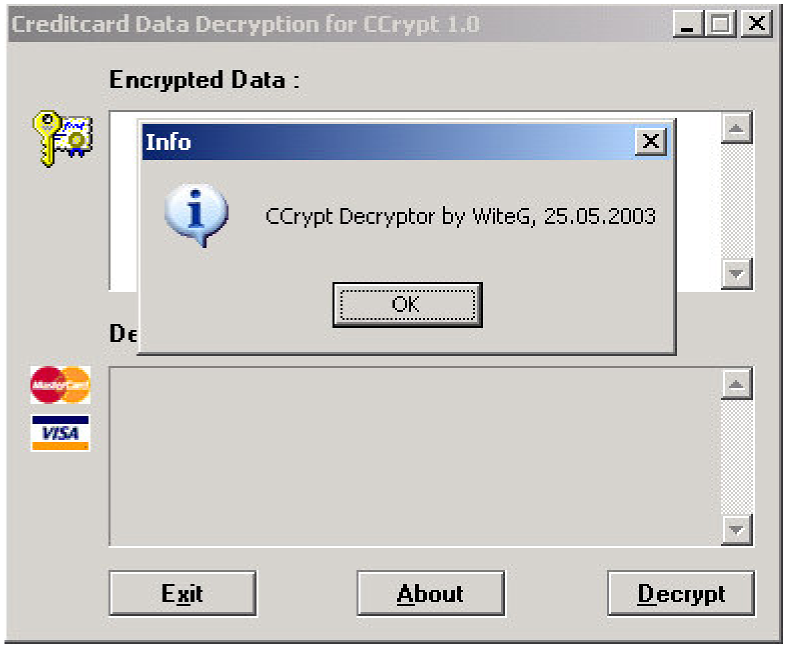
Quoted from the release notes:

“Encoding your credit card data If you feel uncomfortable sending your credit card information over the internet, we suggest to use our tool CCrypt to encrypt your data offline before e-mailing it to us.”

A decryptor was released shortly afterwards

The Requisites

- Key generation


Creditcard Data Decryption for CCrypt 1.0

Encrypted Data :

Info

CCrypt Decryptor by WiteG, 25.05.2003

Flawed by design, Winace's CCrypt was quickly defeated

The Requisites

• Key generation

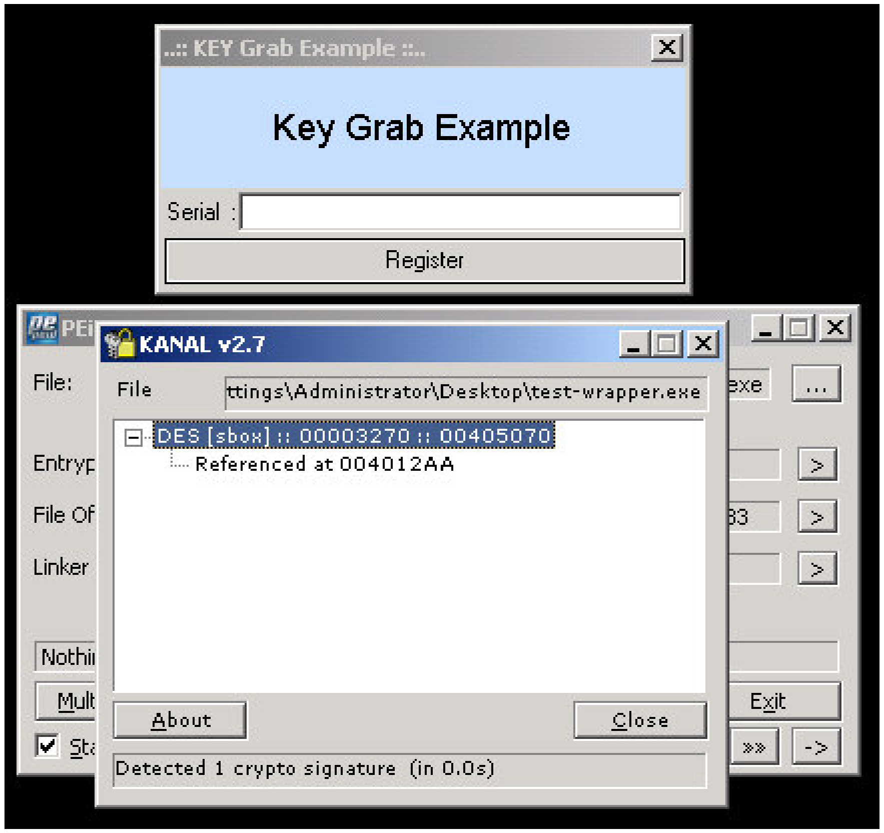
Sadly this isn’t the only instance of bad implementation when it comes to storing/generating keys.

Software registration schemes that rely on symmetric cryptography are often easily defeated. Simple crypto analyzers can detect the cipher in use and thus alert the circumventor of what to expect. This of course gives developers the chance to obfuscate their applications with large prime listings, sbox’s of ciphers not in use and an array of other items a generic crypto analyzer might pick up on.

Asymmetric ciphers normally fair better when put to use in a software registration context, often because of large number libraries that can appear confusing when sifting through disassembled code. Often the complex math involved acts as a decent deterrent.

The Requisites

• Key generation



File ttings\Administrator\Desktop\test-wrapper.exe

```
DES [sbox] :: 00003270 :: 00405070
  Referenced at 004012AA
```

Detected 1 crypto signature (in 0.0s)

PEiD’s built in crypto analyzer detects a DES substitution box resident in our test-wrapper

The Requisites

• Transmission

Pre-secured channels using a strenuously tested and well known protocol such as IPsec are not always an available or feasible option.

What if the data to be sent is strictly confidential and the only transmission medium available is a clear text channel such as SMTP/POP3 or SMB?

What about encapsulating the data to be sent (in this case we'll assume it's a document) in a self-decrypting/extracting executable wrapper?

Software that performs such functions is vast in number, but they often all succumb to the same downfalls…

The Requisites

• Transmission

Software Wrapper: The data is first encrypted, often compressed, and then in most cases bundled inside a binary executable.

Of course it is generally dependant on the software being used, but occasionally it’s no large task separating the encrypted data from the rest of the binary with nothing more than a hex or PE editor at your disposal.

Often it is the case that such programs will append the encryption key (assuming PK/KE isn’t put to use) either to the end of the encrypted data, or stores it permanently within the wrapper software.

Another common mistake is generating a replica of the actual key in memory when checking against an entered key. This of course means the key can be fished from the executable…

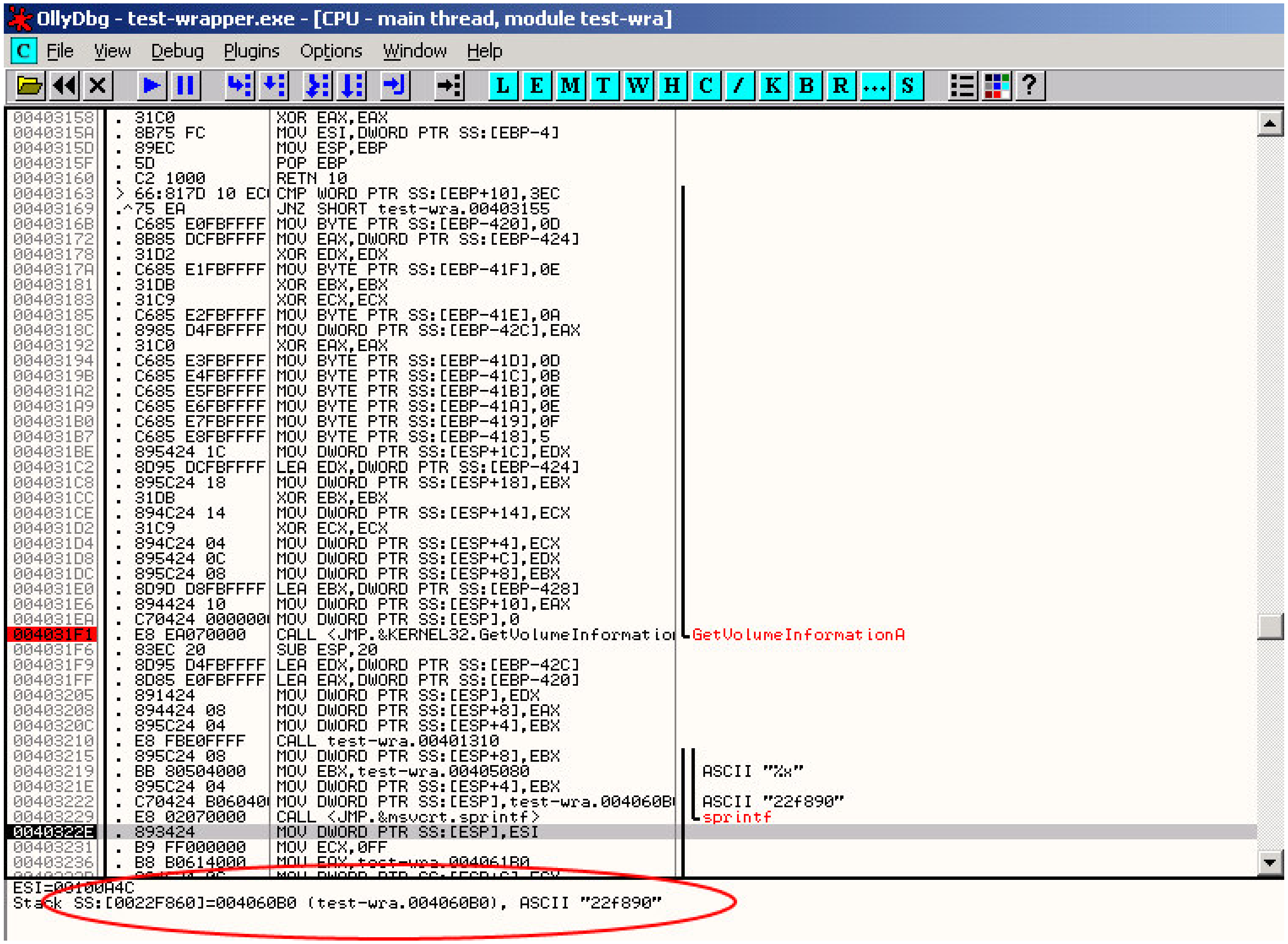
The Requisites

• Transmission

If for example, the wrapper software accepts input from the user, and based upon the input hashes or generates a key which must be identical to the original encryption key.

This allows for a malicious party who has intercepted the wrapper to harvest the key relatively easily.

A better method would be to prompt the recipient for a key, decrypt that input and used the decrypted data as the key. This of course would mean that the actual key would have to be encrypted and communicated to the recipient through other secure channels



```
OllyDbg - test-wrapper.exe - [CPU - main thread, module test-wra]

00403158  . 31C0           XOR EAX,EAX
0040315A  . 8B75 FC        MOV ESI,DWORD PTR SS:[EBP-4]
0040315D  . 89EC           MOV ESP,EBP
0040315F  . 5D             POP EBP
00403160  . C2 1000        RETN 10
00403163  > 66:817D 10 EC  CMP WORD PTR SS:[EBP+10],3EC
00403169  .^75 EA          JNZ SHORT test-wra.00403155
0040316B  . C685 E0FBFFFF  MOV BYTE PTR SS:[EBP-420],0D
00403172  . 8B85 DCFBFFFF  MOV EAX,DWORD PTR SS:[EBP-424]
00403178  . 31D2           XOR EDX,EDX
0040317A  . C685 E1FBFFFF  MOV BYTE PTR SS:[EBP-41F],0E
00403181  . 31DB           XOR EBX,EBX
00403183  . 31C9           XOR ECX,ECX
00403185  . C685 E2FBFFFF  MOV BYTE PTR SS:[EBP-41E],0A
0040318C  . 8985 D4FBFFFF  MOV DWORD PTR SS:[EBP-42C],EAX
00403192  . 31C0           XOR EAX,EAX
00403194  . C685 E3FBFFFF  MOV BYTE PTR SS:[EBP-41D],0D
0040319B  . C685 E4FBFFFF  MOV BYTE PTR SS:[EBP-41C],0B
004031A2  . C685 E5FBFFFF  MOV BYTE PTR SS:[EBP-41B],0E
004031A9  . C685 E6FBFFFF  MOV BYTE PTR SS:[EBP-41A],0E
004031B0  . C685 E7FBFFFF  MOV BYTE PTR SS:[EBP-419],0F
004031B7  . C685 E8FBFFFF  MOV BYTE PTR SS:[EBP-418],5
004031BE  . 895424 1C      MOV DWORD PTR SS:[ESP+1C],EDX
004031C2  . 8D95 DCFBFFFF  LEA EDX,DWORD PTR SS:[EBP-424]
004031C8  . 895C24 18      MOV DWORD PTR SS:[ESP+18],EBX
004031CC  . 31DB           XOR EBX,EBX
004031CE  . 894C24 14      MOV DWORD PTR SS:[ESP+14],ECX
004031D2  . 31C9           XOR ECX,ECX
004031D4  . 894C24 04      MOV DWORD PTR SS:[ESP+4],ECX
004031D8  . 895424 0C      MOV DWORD PTR SS:[ESP+C],EDX
004031DC  . 895C24 08      MOV DWORD PTR SS:[ESP+8],EBX
004031E0  . 8D9D D8FBFFFF  LEA EBX,DWORD PTR SS:[EBP-428]
004031E6  . 894424 10      MOV DWORD PTR SS:[ESP+10],EAX
004031EA  . C70424 000000  MOV DWORD PTR SS:[ESP],0
004031F1  . E8 EA070000    CALL <JMP.&KERNEL32.GetVolumeInformatio  GetVolumeInformationA
004031F6  . 83EC 20        SUB ESP,20
004031F9  . 8D95 D4FBFFFF  LEA EDX,DWORD PTR SS:[EBP-42C]
004031FF  . 8D85 E0FBFFFF  LEA EAX,DWORD PTR SS:[EBP-420]
00403205  . 891424         MOV DWORD PTR SS:[ESP],EDX
00403208  . 894424 08      MOV DWORD PTR SS:[ESP+8],EAX
0040320C  . 895C24 04      MOV DWORD PTR SS:[ESP+4],EBX
00403210  . E8 FBE0FFFF    CALL test-wra.00401310
00403215  . 895C24 08      MOV DWORD PTR SS:[ESP+8],EBX
00403219  . BB 80504000    MOV EBX,test-wra.00405080                 ASCII "%x"
0040321E  . 895C24 04      MOV DWORD PTR SS:[ESP+4],EBX
00403222  . C70424 B06040  MOV DWORD PTR SS:[ESP],test-wra.004060B   ASCII "22f890"
00403229  . E8 02070000    CALL <JMP.&msvcrt.sprintf>                sprintf
0040322E  . 893424         MOV DWORD PTR SS:[ESP],ESI
00403231  . B9 FF000000    MOV ECX,0FF
00403236  . B8 B0614000    MOV EAX,test-wra.004061B0

ESI=00100A4C
Stack SS:[0022F860]=004060B0 (test-wra.004060B0), ASCII "22f890"
```

Avoid re-creating the original key for verification purposes

The Requisites

• Storage

Encrypted data can reside anywhere from a software perspective, so long as it stays encrypted.

If a cryptographic volume or partition is made use of to store incoming data that has been encrypted, then one needn’t worry where data is decrypted to, so long as it is being written to encrypted storage it will be safe, right?

What about the decryption process?

The Requisites

- Storage

In the case of the aforementioned mentioned wrappers, data is placed in memory as it's siphoned through the decryption process, and in the vast majority of instances remains, if at least temporarily, decrypted in memory before being written to permanent storage.

Another viable question to ask is where are keys being stored throughout this process?

What happens when data is flushed to the swap file? Not all OS's make provision for encrypting swap files.

The Requisites

- Storage

Let’s assume that usage of cryptographic storage device is not available, and the user intends to simply view the data after decryption and then delete the decrypted data afterwards, retaining it only in its encrypted form.

However, after deletion, residual data will remain on the disk until it is written over by the OS. Effectively allowing a time frame for malicious parties to recover the decrypted data.

Re-encrypting the data before deletion would prevent this, but might not always be feasible.

Proprietary Algorithms

• Restricted Algorithms

Un-disclosed symmetric algorithms that rely on the secrecy of the algorithm instead of the secrecy of the key can never be considered secure.

If the algorithm is compromised it becomes completely worthless. Vendors who release software claiming to make use of their own algorithms don’t realize how quickly an algorithm can be reverse engineered. An example of this is the proprietary cipher used to encrypt DVD data.

Time-tested ciphers that have been submitted to vast amounts of crypt analysis and have proven themselves strong, should always be selected over unknown and untested ones.

Software Protection Mechanisms

What if cryptography where applied to an online software registration/verification scheme?

A working example of this might be the following scenario:

A software vendor requires users to purchase their product online. They have decided to implement a diffie-hellman key exchange to create valid registration numbers for customers.

They release a trial or feature-disabled version of the product. Customers are able to download and test the product to see if they might want to purchase it.

If they opt to buy the product, they are required to fill out an online billing form.

Software Protection Mechanisms

After completing the form they are presented with 2 different numbers which are to be entered into the products registration form.

Call these numbers p and g.

Where p is a safe prime number and g is primitive mod p

The product then generates its own random number (a) which is transmitted to the vendor's registration repository for storage, and then calculates

Product number = g ^ a % p

(g to the power a mod p)

Software Protection Mechanisms

To confirm payment the product transmits it's product number to the vendor's registration repository.

The vendor then generates it's own random number (b) and calculates:

Vendor number = g ^ b % p

(g to the power b mod p)

The vendor then calculates:

Customer key = vendor number ^ a % p

Vendor key = product number ^ b % p

(vendor key / customer key to the power b / a mod p)

Software Protection Mechanisms

If the vendor key and customer key match the vendor then transmits the original vendor number (Vendor number = g ^ b % p) back to the product.

The product now calculates the customer key once more:

Customer key = vendor number ^ a % p

And uses the result to register the software.

Summary

As technology evolves, seemingly unfeasible attacks such as factoring and brute-forcing will become more and more of a reality.

Cryptographers will have to devise new and cunning ways to harness and implement evolving technologies whilst dispelling new methods of attack.