How to Crack VISA PIN

Have you ever wonder what would happen if you loose your credit or debit card and someone finds it. Would this person be able to withdraw cash from an ATM guessing, somehow, your PIN? Moreover, if you were who finds someone's card would you try to guess the PIN and take the chance to get some easy money? Of course the answer to both questions should be "no". This work does not deal with the second question, it is a matter of personal ethics. Herewith I try to answer the first question.

All the information used for this work is public and can be freely found in Internet. The rest is a matter of mathematics and programming, thus we can learn something and have some fun. I reveal no secrets. Furthermore, the aim (and final conclusion) of this work is to demonstrate that PIN algorithms are still strong enough to provide sufficient security. We all know technology is not the weak point.

This work analyzes one of the most common PIN algorithms, VISA PVV, used by many ATM cards (credit and debit cards) and tries to find out how resistant is to PIN guessing attacks. By "guessing" I do not mean choosing a random PIN and trying it in an ATM. It is well known that generally we are given three consecutive trials to enter the right PIN, if we fail ATM keeps the card. As VISA PIN is four digit long it's easy to deduce that the chance for a random PIN guessing is 3/10000 = 0.0003, it seems low enough to be safe; it means you need to loose your card more than three thousand times (or loosing more than three thousand cards at the same time until there is a reasonable chance of loosing money.

What I really meant by "guessing" was breaking the PIN algorithm so that given any card you can immediately know the associated PIN. Therefore this document studies that possibility, analyzing the algorithm and proposing a method for the attack. Finally we give a tool which implements the attack and present results about the estimated chance to break the system. Note that as long as other banking security related algorithms (other PIN formats such as IBM PIN or card validation signatures such as CVV or CVC) are similar to VISA PIN, the same analysis can be done yielding nearly the same results and conclusions.

VISA PVV algorithm

One of the most common PIN algorithms is the VISA PIN Verification Value (PVV). The customer is given a PIN and a magnetic stripe card. Encoded in the magnetic stripe is a four digit number, called PVV. This number is a cryptographic signature of the PIN and other data related to the card. When a user enters his/her PIN the ATM reads the magnetic stripe, encrypts and sends all this information to a central computer. There a trial PVV is computed using the customer entered PIN and the card information with a cryptographic algorithm. The trial PVV is compared with the PVV stored in the card, if they match the central computer returns to the ATM authorization for the transaction. See in more detail.

The description of the PVV algorithm can be found in two documents linked in the previous page. In summary it consists in the encryption of a 8 byte (64 bit) string of data, called Transformed Security Parameter (TSP), with DES algorithm (DEA) in Electronic Code Book mode (ECB) using a secret 64 bit key. The PVV is derived from the output of the encryption process, which is a 8 byte string. The four digits of the PVV (from left to right) correspond to the first four decimal digits (from left to right) of the output from DES when considered as a 16 hexadecimal character (16 x 4 bit = 64 bit) string. If there are no four decimal digits among the 16 hexadecimal characters then the PVV is completed taken (from left to right) non decimal characters and decimalizing them by using the conversion A->0, B->1, C->2, D->3, E->4, F->5. Here is an example:

Output from DES: 0FAB9CDEFFE7DCBA

PVV: 0975

The strategy of avoiding decimalization by skipping characters until four decimal digits are found (which happens to be nearly all the times as we will see below) is very clever because it avoids an important bias in the distribution of digits which has been proven to be fatal for other systems, although the impact on this system would be much lower. See also a related problem not applying to VISA PVV.

The TSP, seen as a 16 hexadecimal character (64 bit) string, is formed (from left to right) with the 11 rightmost digits of the PAN (card number) excluding the last digit (check digit), one digit from 1 to 6 which selects the secret encrypting key and finally the four digits of the PIN. Here is an example:

PAN: 1234 5678 9012 3445
Key selector: 1
PIN: 2468

TSP: 5678901234412468

Obviously the problem of breaking VISA PIN consists in finding the secret encrypting key for DES. The method for that is to do a brute force search of the key space. Note that this is not the only method, one could try to find a weakness in DEA, many tried, but this old standard is still in wide use (now been replaced by AES and RSA, though). This demonstrates it is robust enough so that brute force is the only viable method (there are some better attacks but not practical in our case, for a summary see LASEC memo and for the dirty details see Biham & Shamir 1990, Biham & Shamir 1991, Matsui 1993, Biham & Biryukov 1994 and Heys 2001).

The key selector digit was very likely introduced to cover the possibility of a key compromise. In that case they just have to issue new cards using another key selector. Older cards can be substituted with new ones or simply the ATM can transparently write a new PVV (corresponding to the new key and keeping the same PIN) next time the customer uses his/her card. For the shake of security all users should be asked to change their PINs, however it would be embarrassing for the bank to explain the reason, so very likely they would not make such request.

Preparing the attack
A brute force attack consists in encrypting a TSP with known PVV using all possible encrypting keys and compare each obtained PVV with the known PVV. When a match is found we have a candidate key. But how many keys we have to try? As we said above the key is 64 bit long, this would mean we have to try 2^64 keys. However this is not true. Actually only 56 HFR are effective in DES keys because one bit (the least significant) out of each octet was historically reserved as a checksum for the others; in practice those 8 HFR (one for each of the 8 octets) are ignored.

Therefore the DES key space consists of 2^56 keys. If we try all these keys will we find one and only one match, corresponding to the bank secret key? Certainly not. We will obtain many matching keys. This is because the PVV is only a small part (one fourth) of the DES output. Furthermore the PVV is degenerated because some of the digits (those between 0 and 5 after the last, seen from left to right, digit between 6 and 9) may come from a decimal digit or from a decimalized hexadecimal digit of the DES output. Thus many keys will produce a DES output which yields to the same matching PVV.

Then what can we do to find the real key among those other false positive keys? Simply we have to encrypt a

second different TSP, also with known PVV, but using only the candidate keys which gave a positive matching with the first TSP-PVV pair. However there is no guarantee we won't get again many false positives along with the true key. If so, we will need a third TSP-PVV pair, repeat the process and so on.

Before we start our attack we have to know how many TSP-PVV pairs we will need. For that we have to calculate the probability for a random DES output to yield a matching PVV just by chance. There are several ways to calculate this number and here I will use a simple approach easy to understand but which requires some background in mathematics of probability.

A probability can always be seen as the ratio of favorable cases to possible cases. In our problem the number of possible cases is given by the permutation of 16 elements (the 0 to F hexadecimal digits) in a group of 16 of them (the 16 hexadecimal digits of the DES output). This is given by 16^16 ~ 1.8 * 10^19 which of course coincides with 2^64 (different numbers of 64 HFR). This set of numbers can be separated into five categories:

1. Those with at least four decimal digits (0 to 9) among the 16 hexadecimal digits (0 to F) of the DES output.
2. Those with exactly only three decimal digits.

3. Those with exactly only two decimal digits.

4. Those with exactly only one decimal digit.
5. Those with no decimal digits (all between A and F).

Let's calculate how many numbers fall in each category. If we label the 16 hexadecimal digits of the DES output as X1 to X16 then we can label the first four decimal digits of any given number of the first category as Xi, Xj, Xk and Xl. The number of different combinations with this profile is given by the product 6 i-1 * 10 * 6j-i-1 * 10 * 6k-j-1 * 10 * 6 l-k-1 * 10 * 1616-l where the 6's come from the number of possibilities for an A to F digit, the 10's come from the possibilities for a 0 to 9 digit, and the 16 comes from the possibilities for a 0 to F digit. Now the total numbers in the first category is simply given by the summation of this product over i, j, k, l from 1 to 16 but with i < j < k < l. If you do some math work you will see this equals to the product of 104/6 with the summation over i from 4 to 16 of (i-1) * (i-2) * (i-3) * 6i-4 * 16 16-i ~ 1.8 * 1019.

Analogously the number of cases in the second category is given by the summation over i, j, k from 1 to 16 with i < j < k of the product 6i-1 * 10 * 6j-i-1 * 10 * 6k-j-1 * 10 * 616-k which you can work it out to be 16!/(3! * (16-13)!) * 103 * 6 13 = 16 * 15 * 14/(3 * 2) * 103 * 613 = 56 * 104 * 613 ~ 7.3 * 1015. Similarly for the third category we have the summation over i, j from 1 to 16 with i < j of 6 i-1 * 10 * 6j-i-1 * 10 * 616-j which equals to 16!/(2! * (16-14)!) * 102 * 614 = 2 * 103 * 615 ~ 9.4 * 1014. Again, for the fourth category we have the summation over i from 1 to 16 of 6i-1 * 10 * 616-i = 160 * 615 ~ 7.5 * 1013. And finally the amount of cases in the fifth category is given by the permutation of six elements (A to F digits) in a group of 16, that is, 616 ~ 2.8 * 1012.

I hope you followed the calculations up to this point, the hard part is done. Now as a proof that everything is right you can sum the number of cases in the 5 categories and see it equals the total number of possible cases we calculated before. Do the operations using 64 bit numbers or rounding (for floats) or overflow (for integers) errors won't let you get the exact result.

Up to now we have calculated the number of possible cases in each of the five categories, but we are interested in obtaining the number of favorable cases instead. It is very easy to derive the latter from the former as this is just fixing the combination of the four decimal digits (or the required hexadecimal digits if there are no four decimal digits) of the PVV instead of letting them free. In practice this means turning the 10's in the formula above into 1's and the required amount of 6's into 1's if there are no four decimal digits. That is, we have to divide the first result by 104, the second one by 103 * 6, the third one by 102 * 62 , the fourth one by 10 * 63 and the fifth one by 64 . Then the number of favorable cases in the five categories are approximately 1.8 * 1015, 1.2 * 1012, 2.6 * 1011 , 3.5 * 1010, 2.2 * 109 respectively.

Now we are able to obtain what is the probability for a DES output to match a PVV by chance. We just have to add the five numbers of favorable cases and divide it by the total number of possible cases. Doing this we obtain that the probability is very approximately 0.0001 or one out of ten thousand. Is it strange this well rounded result? Not at all, just have a look at the numbers we calculated above. The first category dominates by several orders of magnitude the number of favorable and possible cases. This is rather intuitive as it seems clear that it is very unlikely not having four decimal digits (10 chances out of 16 per digit) among 16 hexadecimal digits. We saw previously that the relationship between the number of possible and favorable cases in the first category was a division by 10^4, that's where our result p = 0.0001 comes from.

Our aim for all these calculations was to find out how many TSP-PVV pairs we need to carry a successful brute force attack. Now we are able to calculate the expected number of false positives in a first search: it will be the number of trials times the probability for a single random false positive, i.e. t * p where t = 2^56, the size of the key space. This amounts to approximately 7.2 * 10^12, a rather big number. The expected number of false positives in the second search (restricted to the positive keys found in the first search) will be (t * p) * p, for a third search will be ((t * p) * p) * p and so on. Thus for n searches the expected number of false positives will be t * p^n.

We can obtain the number of searches required to expect just one false positive by expressing the equation t * p^n = 1 and solving for n. So n equals to the logarithm in base p of 1/t, which by properties of logarithms it yields n = log(1/t)/log(p) ~ 4.2. Since we cannot do a fractional search it is convenient to round up this number. Therefore what is the expected number of false positives if we perform five searches? It is t * p^5 ~ 0.0007 or approximately 1 out of 1400. Thus using five TSP-PVV pairs is safe to obtain the true secret key with no false positives.

The attack

Once we know we need five TSP-PVV pairs, how do we get them? Of course we need at least one card with known PIN, and due to the nature of the PVV algorithm, that's the only thing we need. With other PIN systems, such as IBM, we would need five cards, however this is not necessary with VISA PVV algorithm. We just have to read the magnetic stripe and then change the PIN four times but reading the card after each change.

It is necessary to read the magnetic stripe of the card to get the PVV and the encrypting key selector. You can buy a commercial magnetic stripe reader or make one yourself following the instructions you can find in the previous page and links therein. Once you have a reader see this description of standard magnetic tracks to find out how to get the PVV from the data read. In that document the PVV field in tracks 1 and 2 is said to be five character long, but actually the true PVV consists of the last four digits. The first of the five digits is the key selector. I have only seen cards with a value of 1 in this digit, which is consistent with the standard and with the secret key never being compromised (and therefore they did not need to move to another key changing the selector).

I did a simple C program, getpvvkey.c, to perform the attack. It consists of a loop to try all possible keys to encrypt the first TSP, if the derived PVV matches the true PVV a new TSP is tried, and so on until there is a mismatch, in which case the key is discarded and a new one is tried, or the five derived PVVs match the corresponding true PVVs, in which case we can assume we got the bank secret key, however the loop goes on until it exhausts the key space. This is done to assure we find the true key because there is a chance (although very low) the first key found is a false positive.

It is expected the program would take a very long time to finish and to minimize the risks of a power cut, computer hang out, etc. it does checkpoints into the file getpvvkey.dat from time to time (the exact time depends on the speed of the computer, it's around one hour for the fastest computers now in use). For the same reason if a positive key is found it is written on the file getpvvkey.key. The program only displays one message at the beginning, the starting position taken from the checkpoint file if any, after that nothing more is displayed.

The DES algorithm is a key point in the program, it is therefore very important to optimize its speed. I tested several implementations: libdes, SSLeay, openssl, cryptlib, nss, libgcrypt, catacomb, libtomcrypt, cryptopp, ufccrypt. The DES functions of the first four are based on the same code by Eric Young and is the one which performed best (includes optimized C and x86 assembler code). Thus I chose libdes which was the original implementation and condensed all relevant code in the files encrypt.c (C version) and x86encrypt.s (x86 assembler version). The code is slightly modified to achieve some enhancements in a brute force attack: the initial permutation is a fixed common steep in each TSP encryption and therefore can be made just one time at the beginning. Another improvement is that I wrote a completely new setkey function (I called it nextkey) which is optimum for a brute force loop.

To get the program working you just have to type in the corresponding place five TSPs and their PVVs and then compile it. I have tested it only in UNIX platforms, using the makefile Makegetpvvkey to compile (use the command "make -f Makegetpvvkey"). It may compile on other systems but you may need to fix some things. Be sure that the definition of the type long64 corresponds to a 64 bit integer. In principle there is no dependence on the endianness of the processor. I have successfully compiled and run it on Pentium-Linux, Alpha-Tru64, Mips-Irix and Sparc-Solaris. If you do not have and do not want to install Linux (you don't know what you are missing ;-) you still have the choice to run Linux on CD and use my program, see my page running Linux without installing it.

Once you have found the secret bank key if you want to find the PIN of an arbitrary card you just have to write a similar program (sorry I have not written it, I'm too lazy that would try all 10^4 PINs by generating the corresponding TSP, encrypting it with the (no longer) secret key, deriving the PVV and comparing it with the PVV in the magnetic stripe of the card. You will get one match for the true PIN. Only one match? Remember what we saw above, we have a chance of 0.0001 that a random encryption matches the PVV. We are trying 10000 PINs (and therefore TSPs) thus we expect 10000 * 0.0001 = 1 false positive on average.

This is a very interesting result, it means that, on average, each card has two valid PINs: the customer PIN and the expected false positive. I call it "false" but note that as long as it generates the true PVV it is a PIN as valid as the customer's one. Furthermore, there is no way to know which is which, even for the ATM; only customer knows. Even if the false positive were not valid as PIN, you still have three trials at the ATM anyway, enough on average.

Therefore the probability we calculated at the beginning of this document about random guessing of the PIN has to be corrected. Actually it is twice that value, i.e., it is 0.0006 or one out of more than 1600, still safely low.

Results

It is important to optimize the compilation of the program and to run it in the fastest possible processor due to the long expected run time. I found that the compiler optimization flag -O gets the better performance, thought some improvement is achieved adding the -fomit-frame-pointer flag on Pentium-Linux, the -spike flag on Alpha-Tru64, the -IPA flag on Mips-Irix and the -fast flag on Sparc-Solaris. Special flags (-DDES_PTR -DDES_RISC1 DDES_RISC2 -DDES_UNROLL -DASM) for the DES code have generally benefits as well. All these flags have already been tested and I chose the best combination for each processor (see makefile) but you can try to fine tune other flags.

According to my tests the best performance is achieved with the AMD Athlon 1600 MHz processor, exceeding 3.4 million keys per second. Interestingly it gets better results than Intel Pentium IV 1800 MHz and 2000 MHz (see figures below, click on them to enlarge). I believe this is due to some I/O saturation, surely cache or memory access, that the AMD processor (which has half the cache of the Pentium) or the motherboard in which it is running, manages to avoid. In the first figure below you can see that the DES breaking speed of all processors has more or less a linear relationship with the processor speed, except for the two Intel Pentium I mentioned before. This is logical, it means that for a double processor speed you'll get double breaking speed, but watch out for saturation effects, in this case it is better the AMD Athlon 1600 MHz, which will be even cheaper than the Intel Pentium 1800 MHz or 2000 MHz.

In the second figure we can see in more detail what we would call intrinsic DES break power of the processor. I get this value simply dividing the break speed by the processor speed, that is, we get the number of DES keys tried per second and per MHz. This is a measure of the performance of the processor type independently of its speed. The results show that the best processor for this task is the AMD Athlon, then comes the Alpha and very close after it is the Intel Pentium (except for the higher speed ones which perform very poor due to the saturation effect). Next is the Mips processor and in the last place is the Sparc. Some Alpha and Mips processors are located at bottom of scale because they are early releases not including enhancements of late versions. Note that I included the performance of x86 processors for C and assembler code as there is a big difference. It seems that gcc is not a good generator of optimized machine code, but of course we don't know whether a manual optimization of assembler code for the other processors (Alpha, Mips, Sparc) would boost their results compared to the native C compilers (I did not use gcc for these other platforms) as it happens with the x86 processor.

The top mark I got running my program was approximately 3 423 922 keys/second using the AMD processor. So, how much time would need the AMD to break the VISA PIN? It would simply be the ratio between the size of the key space and the key trying rate, that is, 2^56 keys/3 423 922 keys/second ~ 2.1 * 10^10 seconds ~ 244 thousand days ~ 667 years. This is the time for the program to finish, but on average the true secret key will be found by half that time. Using commercial cryptographic cards (like the IBM PCI Cryptographic Coprocessor or the XL-Crypt Encryption Accelerator) does not help very much, they are, at most, 2 times faster than my top mark, i.e. it would take more than a hundred years to find the key, at best. Some more speed might be achieved (double, at most) by using a dedicated gigabit VPN box or similar hardware in a way surely not foreseen by the manufacturer ;-)

Even if you manage to get a hundred newest AMD or Pentium processors working in parallel it would still take more than 3 years to find the key (if they are provided with crypto-cards the time might be reduced to less than two years or to less than one year in case of a hundred gigabit VPN boxes). It is clear that only expensive dedicated hardware (affordable only by big institutions) or a massive Internet cooperative attack would success in a reasonable time (both things were already made). These are the good news. The bad news is that I have deliberately lied a little bit (you may already noticed it): VISA PVV algorithm allows for the use of triple DES (3DES) encryption using a 128 bit (only 112 effective) encrypting key. If 3-DES is indeed in use by the PVV system you can still use the same attack but you would need four additional TSP-PVV pairs (no problem with that) and it would take more than 3 * 2^56 times more to find the double length key. Forget it.

PVV algorithm with triple DES consists in the encryption of the TSP with the left half of the encrypting key, then it decrypts the result with the right half of the key and encrypts the result again with the left half of the key. Note that if you use a symmetric 128 bit key, that is, the left half equals the right half, you get a single DES encryption with a single 64 bit key. In this case the algorithm degenerates into the one I explained above. That's why I did this work, because PVV system is old and maybe when it was implanted 3-DES was not viable (due to hardware limitations)

or it seemed excessive (by that time) to the people responsible of the implementation, so that it might be possible some banks are using the PVV algorithm with single DES encryption.

Finally we can conclude that the VISA PVV algorithm as in its general form using 3-DES is rather secure. It may only be broken using specially designed hardware (implying an enormous inversion and thus not worth, see Wayner and Wiener) which would exceed the encryption rate of the newest processors by many orders of magnitude. However the apparently endless exponential growing of the computer capacities as well as that of the Internet community makes to think that PVV system might be in real danger within a few years. Of course those banks using PVV with single DES (if any) are already under true risk of an Internet cooperative attack. You might believe that is something very hard to coordinate, I mean convincing people, but think about trojan and virus programs and you will see it is not so difficult to carry on.