How we Learned to Cheat in Online Poker: A Study in Software Security

Brad Arkin, Frank Hill, Scott Marks, Matt Schmid, Thomas John Walls, and Gary McGraw
Reliable Software Technologies
Software Security Group

Poker is card game that many people around the world enjoy. Poker is played at kitchen tables, in casinos and cardrooms, and, recently, on the Web. A few of us here at Reliable Software Technologies (http://www.rstcorp.com) play poker. Since many of us spend a good amount of time on-line, it was only a matter of time before some of us put the two interests together. This is the story of how our interests in online poker and software security mixed to create a spectacular security exploit.

The PlanetPoker Internet cardroom (http://www.planetpoker.com) offers real-time Texas Hold'em games against other people on the Web for real money. Being software professionals who help companies deliver secure, reliable, and robust software, we were curious about the software behind the online game. How did it work? Was it fair? An examination of the FAQs at PlanetPoker, including the shuffling algorithm (which was ironically published to help demonstrate the game's integrity), was enough to start our analysis wheels rolling. As soon as we saw the shuffling algorithm, we began to suspect there might be a problem. A little investigation proved this intuition correct.

The Game

In Texas Hold'em, each player is dealt two cards (called the *pocket cards*). The initial deal is followed by a round of betting. After the first round of betting, all remaining cards are dealt face up and are shared by all players. The dealer places three cards face up on the board (called the *flop*). A second round of betting takes place. Texas Hold'em is usually a fixed limit game, meaning that there are fixed amounts that a player may bet in each betting round. For example, in a $3-$6 game, the first two betting rounds are three dollar bets while the third and fourth betting rounds are six dollar bets. After the second round of betting, the dealer places another card face up on the board (called the *turn*). A third round of betting takes place. Finally, the dealer places the last card face up on the board (called the *river*), and a final round of betting takes place. Each remaining player takes their two pocket cards and combines them with the five community cards to make the best five card poker hand. The best hand among the players is determined by standard poker hand order.

Texas Hold'em is a fast-paced and exciting game. Bluffing is an essential part of the game, and quick decisions about who is holding what sorts of cards separate winners from losers. Interestingly, Texas Hold'em is the poker game played at the World Series of Poker which is held annually in Las Vegas.

Now that everybody and their dog is online, and virtually all types of businesses are represented on the Internet, it's only natural that Internet casinos and cardrooms are too. Even with the reasonably easy availability of casinos on Indian reservations and riverboats, there is still real demand for more accessible games that are easy to join. Being able to play on-line in the comfort of your own home (not to mention in your pajamas), without having to endure second hand smoke and obnoxious players, is definitely appealing.

Security Risks Abound

All this convenience comes at a price. Unfortunately, there are real risks to playing poker on-line. The casino may be a fraud, existing only to take money from naïve players without ever intending to pay back winnings. The server running the on-line casino could be cracked by a malicious attacker looking for credit card numbers, or trying to leverage some advantage in the game. Since a majority of casinos don’t authenticate or encrypt the network traffic between the player running the client program and the server hosting the card game, a malicious player could conceivably examine the network traffic (with a classic

person-in-the-middle attack) for the purposes of determining his opponent's cards. These risks are all very familiar to Internet security experts.

Collusion is a problem that is unique to poker (as opposed to other games like blackjack or craps), since poker players play against each other and not the casino itself. Collusion occurs when two or more players seated at the same table work together as a team, often using the same bankroll. Colluding players know what their team members' hands are (often through subtle signals), and bet with the purpose of maximizing their team's profits on any given hand. Though collusion is a problem in real cardrooms, it is a much more serious problem for online poker. Using tools like instant messaging and telephone conference calls makes collusion a serious risk to online poker players. What if all the players in an online game are all cooperating to bilk an unsuspecting Web patsy? How can you be assured that you're never a victim of this attack?

Last, but not least (especially in terms of our story), there is a real risk that the software behind an online poker game may be flawed. Software problems are a notorious form of security risk often overlooked by companies obsessed with firewalls and cryptography. The problem is that a software application can introduce truck-sized security holes into a system. We spend a great deal of time in our day jobs finding and solving software security problems for our clients. It is only natural that we turned our attention to online poker. The rest of this article is devoted to a discussion of software security problems we found in a popular online poker game.

Software Security Risks

Shuffling a Virtual Deck of Cards

The first software flaw we'll focus on involves shuffling virtual cards. What does it mean to shuffle a deck of cards fairly? Essentially, every possible combination of cards should have an equal likelihood of appearing. We'll call each such ordering of the 52 cards a *shuffle*.

In a real deck of cards, there are 52! (approximately 2^{226}) possible unique shuffles. When a computer shuffles a virtual deck of cards, it selects one of these possible combinations. There are many algorithms that can be used to shuffle a deck of cards, some of which are better than others (and some of which are just plain wrong).

We found that the algorithm used by ASF Software, Inc. (http://www.asfgames.com), the company that produces the software used by most of the online poker games, suffered from many flaws. ASF has changed their algorithm since we contacted them regarding our discovery. We have not looked at their new approach. Getting everything exactly right from a security perspective is not easy (as the rest of this article will show).

Figure 1: The Flawed ASF Shuffling Algorithm

```
procedure TDeck.Shuffle;
var
        ctr: Byte;
        tmp: Byte;

        random_number: Byte;
begin
        { Fill the deck with unique cards }
        for ctr := 1 to 52 do
                Card[ctr] := ctr;

        { Generate a new seed based on the system clock }
        randomize;
```

```
        { Randomly rearrange each card }
        for ctr := 1 to 52 do begin
                random_number := random(51)+1;
                tmp := card[random_number];
                card[random_number] := card[ctr];
                card[ctr] := tmp;
        end;

        CurrentCard := 1;
        JustShuffled := True;
end;
```

The shuffling algorithm shown in Figure 1 was posted by ASF Software in order to convince people that their computer-generated shuffles were entirely fair. Ironically, it had the exact opposite effect on us.

The algorithm starts by initializing an array with values in order from 1 to 52, representing the 52 possible cards. Then, the program initializes a pseudo-random number generator using the system clock with a call to Randomize(). The actual shuffle is performed by swapping every position in the array, in turn, with a randomly chosen position. The position to swap with is chosen by calls to the pseudo-random number generator.

Problem one: An off-by-one error

Astute programmers will have noticed that the algorithm in question contains an off-by-one error. The algorithm is supposed to traverse the initial deck while swapping each card with any other card. Unlike most Pascal functions, the function Random(n) actually returns a number between 0 and n-1 instead of a number between 1 and n. The algorithm uses the following snippet of code to choose which card to swap with the current card: <random_number := random(51)+1;>. The formula sets random_number to a value between 1 and 51. In short, the algorithm in question never chooses to swap the current card with the last card. When ctr finally reaches the last card, 52, that card is swapped with any other card except itself. That means this shuffling algorithm never allows the 52nd card to end up in the 52nd place. This is an obvious, but easily correctable, violation of fairness.

Problem two: Bad distribution of shuffles

A closer examination of the shuffling algorithm reveals that, regardless of the off-by-one problem, it doesn't return an even distribution of decks. The basic algorithm at the heart of the shuffle is shown in Figure 2.

Shuffling

A closer examination of the algorithm reveals that, regardless of the off-by-one error, it doesn't return an even distribution of shuffles. That is, some shuffles are more likely to be produced than others are. This uneven distribution can be leveraged into an advantage if a tipped-off player is willing to sit at the table long enough.

To illustrate this problem using a small example, we'll shuffle a deck consisting of only three cards (i.e, n=3) using the algorithm described above.

Figure 2: How not to shuffle cards

```
for (i is 1 to n)
  Swap i with random position between 1 and n
```

Figure 2 contains the algorithm we used to shuffle our deck of three cards, and also depicts the tree of all possible decks using this shuffling algorithm. If our random number source is a good one, then each leaf on the tree in Figure 2 has an equal probability of being produced.

Given even this small example, you can see that the algorithm does not produce shuffles with equal probability. It will produce the decks 231, 213, and 132 more often than the decks 312, 321, 123. If you were betting on the first card and you knew about these probabilities, you would know that card 2 is more likely to appear than any other card. The uneven probabilities become increasingly exaggerated as the number of cards in the deck increase. When a full deck of 52 cards is shuffled using the algorithm listed above (n=52), the unequal distribution of decks skews the probabilities of certain hands and changes the betting odds. Experienced poker players (who play the odds as a normal course of business) can take advantage of the skewed probabilities.

Figure 3: How to shuffle cards

```
for (i is 1 to 3)
  Swap i with random position between i and 3
```

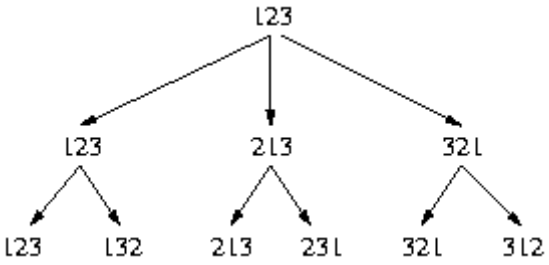


Figure 3 provides a much better shuffling algorithm. The crucial difference between the two algorithms is that number of possible swap positions decreases as you progress through the deck. Once again, we show a tree illustrating this algorithm on our sample deck of three cards. The change between this new algorithm and the one used by ASF is that each card i is swapped with a card from the range [i, n], not [1, n]. This reduces the number of leaves from the $3^3 = 27$ given by the bad algorithm listed above to 3! = 6. The change is important because the n! number of unique leaves means that the new shuffling algorithm generates each possible deck *only once*. Notice that each possible shuffle is produced once and only once so that each deck has an equal probability of occurring. Now that's fair!

Generating Random Numbers on a Deterministic Machine

The first set of software flaws we discussed merely changes the probabilities that certain cards will come up. The associated skews can be used by a clever gambler to gain an edge, but the flaws really don't constitute a complete break in the system. By contrast, the third flaw, which we explain in this section, is a doozy that allows online poker to be completely compromised. A short tutorial on psuedo-random number generators sets the stage for the rest of our story.

How psuedo-random number generators work

Suppose we want to generate a random number between 1 and 52, where every number has an equal probability of appearing. Ideally, we would generate a value on the range from 0 to 1 where every value will occur with equal probability, regardless of the previous value, then multiply that value by 52. Note that there are an infinite number of values between 0 and 1. Also note that computers do not offer infinite precision!

In order to program a computer to do something like the algorithm presented above, a psuedo-random number generator typically produces an integer on the range from 0 to N and returns that number divided by N. The resulting number is always between 0 and 1. Subsequent calls to the generator take the integer result from the first run and pass it through a function to produce a new integer between 0 and N, then return the new integer divided by N. This means the number of unique values returned by any pseudo-random number generator is limited by number of integers between 0 and N. In most common random number generators, N is 2^{32} (approximately 4 billion) which is the largest value that will fit into a 32-bit number. Put another way, there are at most 4 billion possible values produced by this sort of number generator. To tip our hand a bit, this 4 billion number is not all that large.

A number known as the *seed* is provided to a psuedo-random generator as an initial integer to pass through the function. The seed is used to get the ball rolling. Notice that there is *nothing unpredicatable* about the output of a pseudo-random generator. Each value returned by a pseudo-random number generator is completely determined by the previous value it returned (and ultimately, the seed that started it all). If we know the integer used to compute any one value then we know every subsequent value returned from the generator.

The pseudo-random number generator distributed with Borland compilers makes a good example and is reproduced in Figure 4. If we know that the current value of RandSeed is 12345, then the next integer produced will be 1655067934 and the value returned will be 20. The same thing happens every time (which should not be surprising to anyone since computers are completely deterministic).

Figure 4 : Borland's implementation of Random()

```
long long RandSeed = #### ;

unsigned long Random(long max)
{
 long long x ;
 double i ;
 unsigned long final ;
 x = 0xffffffff;
 x += 1 ;

 RandSeed *= ((long long)134775813);
 RandSeed += 1 ;
 RandSeed = RandSeed % x ;
 i = ((double)RandSeed) / (double)0xffffffff ;
 final = (long) (max * i) ;

 return (unsigned long)final;
}
```

Based on historical precedent, seeds for number generators are usually produced based on the system clock. The idea is to use some aspect of system time as the seed. This implies if you can figure out what time a generator is seeded, you will know every value produced by the generator (including what order numbers will appear in). The upshot of all this is that there is nothing unpredictable about psuedo-random numbers. Needless to say, this fact has a profound impact on shuffling algorithms!

On to poker, or how to use a random number generator badly

The shuffling algorithm used in the ASF software always starts with an ordered deck of cards, and then generates a sequence of random numbers used to re-order the deck. Recall that in a real deck of cards, there are 52! (approximately 2^{226}) possible unique shuffles. Also recall that the seed for a 32-bit random number generator must be a 32-bit number, meaning that there are just over 4 billion possible seeds. Since the deck is reinitialized and the generator re-seeded before each shuffle, only 4 billion possible shuffles can result from this algorithm. Four billion possible shuffles is alarmingly less than 52!.

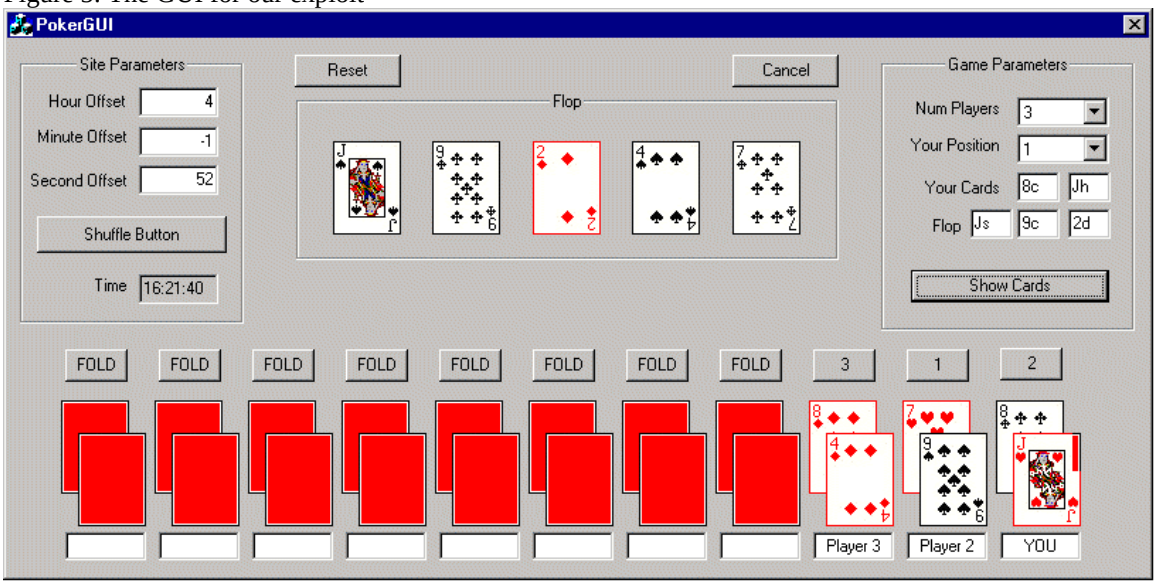
To make matters worse, the algorithm of Figure 1 chooses the seed for the random number generator using the Pascal function Randomize(). This particular Randomize() function chooses a seed based on the number of milliseconds since midnight. There are a mere 86,400,000 milliseconds in a day. Since this number was being used as the seed for the random number generator, the number of possible decks now reduces to 86,400,000. Eight-six million is alarmingly less than four billion. But that's not all. It gets worse.

Breaking the System

The system clock seed gave us an idea that reduced the number of possible shuffles even further. By synchronizing our program with the system clock on the server generating the pseudo-random number, we are able to reduce the number of possible combinations down to a number on the order of 200,000 possibilities. After that move, the system is ours, since searching through this tiny set of shuffles is trivial and can be done on a PC in real time.

The RST exploit itself requires five cards from the deck to be known. Based on the five known cards, our program searches through the few hundred thousand possible shuffles and deduces which one is a perfect match. In the case of Texas Hold'em poker, this means our program takes as input the two cards that the cheating player is dealt, plus the first three community cards that are dealt face up (the flop). These five cards are known after the first of four rounds of betting and are enough for us to determine (in real time, during play) the exact shuffle. Figure 5 shows the GUI we slapped on our exploit. The "Site Parameters" box in the upper left is used to synchronize the clocks. The "Game Parameters" box in the upper right is used to enter the five cards and initiate the search. Figure 5 is a screen shot taken after all cards have been determined by our program. We know who holds what cards, what the rest of the flop looks, and who is going to win *like in advance*.

Figure 5: The GUI for our exploit



Once it knows the five cards, our program generates shuffles until it discovers the shuffle that contains the five cards in the proper order. Since the Randomize() function is based on the server's system time, it is not very difficult to guess a starting seed with a reasonable degree of accuracy. (The closer you get, the fewer possible shuffles you have to look through.) Here's the kicker though; after finding a correct seed once, it is possible to synchronize our exploit program with the server to within a few seconds. This post facto synchronization allows our program to determine the seed being used by the random number generator, and to identify the shuffle being used during all future games in *under one second*!

Technical detail aside, our exploit garnered spectacular press coverage. The coverage emphasizes the human side of our discovery. See our Web site for our original press release, the CNN video clip, and a New York Times story (http://www.rstcorp.com).

Doing Things Properly, or How to Shuffle Virtual Cards

As we have shown, shuffling virtual cards isn't as easy as it may appear at first blush. The best way to go about creating a shuffling algorithm is to develop a technique that can securely produce a well-shuffled deck of cards by relying on sound mathematics. Furthermore, we believe that publishing a good algorithm and opening it up to real-world scrutiny is a good idea (which meshes nicely with the opinions of the Open Source zealots). The main thing here is not relying on security by obscurity. Publishing a bad algorithm (like AFS did) is a bad idea, but so is not publishing a bad algorithm!

Cryptography relies on solid mathematics, not obscurity, to develop strong algorithms used to protect individual, government, and commercial secrets. We think shuffling is similar. We can stretch the analogy to include a parallel between cryptographic key length (which is directly proportional to the strength of many cryptographic algorithms) and the size of the random seed that is used to produce a shuffled deck of cards.

Developing a card-shuffling algorithm is a fairly straightforward task. The first thing to realize is that an algorithm capable of producing each of the 52! shuffles is not really required. The reasoning underlying this claim is that only an infinitesimally small percent of the 52! shuffles will ever be used during play. It is important, however, that the shuffles the algorithm produces maintain an even distribution of cards. A good distribution ensures that each position in the shuffle has an approximately equal chance of holding any one particular card. The distribution requirement is relatively easy to achieve and verify. The following pseudo-code gives a simple card-shuffling algorithm that, when paired with the right random number generator, produces decks of cards with an even distribution.

```
START WITH FRESH DECK
GET RANDOM SEED
FOR CT = 1, WHILE CT <= 52, DO
        X = RANDOM NUMBER BETWEEN CT AND 52 INCLUSIVE
        SWAP DECK[CT] WITH DECK[X]
```

Key to the success of our algorithm is the choice of a random number generator (RNG). The RNG has a direct impact on whether the algorithm above will successfully produce decks of even distribution as well as whether these decks will be useful for secure online card play. To begin with, the RNG itself must produce an even distribution of random numbers. Pseudo-random number generators (PRNG), such as those based on the Lehmer algorithm, have been shown to possess this mathematical property. It is therefore sufficient to use a good PRNG to produce "random" numbers for card shuffling.

As we have seen, choice of initial seed for the PRNG is a make or break proposition. Everything boils down to the seed. It's absolutely essential that players using a deck of cards generated using a PRNG can't determine the seed used to produce that particular shuffle.

A brute force attempt to determine the seed used to produce a particular shuffle can be made by systematically going through each of the possible seeds, producing the associated shuffle, and comparing the result against the deck you're searching for. To avoid susceptibility to this kind of attack, the number of possible seeds needs to be large enough that it is computationally infeasible to perform an exhaustive search within certain time constraints. Note that on average only half of the seed space will need to be searched until a match is found. For the purposes of an online card game, the time constraint would be the length of that game, which is usually on the order of minutes.

In our experience, a simple program running on a Pentium 400 computer is able to examine approximately 2 million seeds per minute. At this rate, this single machine could exhaustively search a 32-bit seed space (2^{32} possible seeds) in a little over a day. Although that time period is certainly beyond the time constraints we have imposed on ourselves, it is certainly not infeasible to use a network of computers to perform a distributed search within our real time bounds.

The subject of brute force attacks serves to emphasize the aptness of our analogy between key length in a cryptographic algorithm and the seed behind shuffling. A brute-force cryptographic attack involves attempting every possible key in order to decrypt a secret message. Likewise a brute-force attack against a shuffling algorithm involves examining every possible seed. A significant body of research studying the necessary lengths of cryptographic keys already exists. Generally speaking, things look like this:

Algorithm	Weak Key	Typical Key	Strong Key
DES	40 or 56	56	Triple-DES
RC4	60	80	128
RSA	512	768 or 1024	2048
ECC	125	170	230

People used to think that cracking 56-bit DES in real time would take too long to be feasible, but history has shown otherwise. In January 1997, a secret DES key was recovered in 96 days. Later efforts broke keys in 41 days, then 56 hours, and, in January 1999, in 22 hours and 15 minutes. The leap forward in cracking ability doesn't bode well for small key lengths or small sets of seeds!

People have even gone so far as to invent special machines to crack cryptographic algorithms. In 1998, the EFF created a special-purpose machine to crack DES messages. The purpose of the machine was to emphasize just how vulnerable DES (a popular, government-sanction algorithm) really is. (For more on the DES cracker, see http://www.eff.org/descracker/.) The ease of "breaking" DES is directly related to the length of its key. Special machines to uncover RNG seeds are not outside the realm of possibility.

We believe that a 32-bit seed space is not sufficient to resist a determined brute-force attack. On the other hand, a 64-bit seed should be resistant to almost any brute force attack. A 64-bit seed makes sense since many computers provide the ability to work with 64-bit integers off the shelf, and a 64-bit number should be sufficient for the purposes of protecting card shuffling against brute-force attacks.

64-bits alone won't do it though. We can't overemphasize that there must be absolutely no way for an attacker to predict or approximate the seed that is used by the PRNG. If there is a way to predict the seed, then the computationally-taxing brute-force attack outlined above becomes irrelevant (since the entire system breaks much more easily). In terms of our exploit, not only did ASF's flawed approach rely on a too-small 32-bit PRNG, but the approach relies on a seed based on the time of day as well. As our exploit demonstrated, there is very little randomness in such an approach.

In final analysis, the security of the entire system relies on picking a random seed in a non-predictable manner. The best techniques for choosing such a random number are hardware-based. Hardware-based approaches rely on unpredictably random data gathered directly from the physical environment. Since online poker and other games involving real money are extremely security critical undertakings, it's probably worth investing the necessary resources to ensure that random number generation is done correctly.

In concert, a good shuffling algorithm and a 64-bit pseudo-random number generator seeded with a proven hardware device should produce shuffles that are both fair and secure. Implementing a fair system is not overly difficult. Online poker players should demand it.

Take Home Message

The broad take home message from this work is simple: when software misbehaves, bad things can happen. Our mission in Reliable Software Technologies' Software Security Group is to stamp out insecure code before it is placed in service. Although we never made use of these potential problems in order to make money, this entire episode should serve as a warning that developing security critical code is never as easy as it sounds. Online poker aside, customers, entrepreneurs, and developers need to be aware of the potential security pitfalls facing any network-based software. It is much better to nip these problems in the bud, during design and risk analysis than to be confronted with a publicly hacked system.

Our story has presented a real world example of an application that really must be secure. Unfortunately, because of the possibly-surprising difficulty of the shuffling problem, the developers failed to deliver a fair and secure online poker solution. Don't let this happen to your software.