

Initial Cryptanalysis of the RSA SecurID Algorithm

Mudge and Kingpin
{mudge,kingpin}@atstake.com

@stake
<http://www.atstake.com>

January 2001

Abstract

Recently, I.C. Wiener published a reverse engineering effort of the RSA SecurID¹ algorithm [1]. There were few speculations on the security ramifications of the algorithm in I.C. Wiener's posting, so this note is an effort to touch upon areas of concern. We have verified that I.C. Wiener's released version of the proprietary algorithm is accurate by comparing it with our own prior reverse engineering of the same algorithm.

Due to the time sensitivity imposed by the public release of RSA's proprietary algorithm, we felt it necessary to release this brief to help people better understand and work toward reducing the risks to which they might currently be exposed. The risk profile of token devices changes when they are implemented in an uncontrolled environment, such as the Internet, and the research in this paper aims to educate and to help manage those risks. The primary concern is the possibility to generate a complete cycle of tokencode outputs given a known secret, which is equivalent to the cloning of a token device.

This short paper will examine several discovered statistical irregularities in functions used within the SecurID algorithm: the time computation and final conversion routines. Where and how these irregularities can be mitigated by usage and policy are explored. We are planning for the release of a more thorough analysis in the near future. This paper does not present methods of determining the secret component by viewing previously generated or successive tokencodes.

¹SecurID is a registered trademark of RSA Security, Inc.

1 Introduction

The RSA SecurID token is currently based upon a proprietary algorithm and provides a 6- to 8-digit tokencode as output. This output is said to be a “new, unpredictable code” [2] displayed at 30- or 60-second intervals. The algorithm being used as of this writing was originally designed for and used on a custom 4-bit microcontroller with an operating speed of less than 1 megahertz. Given these operational and computational capabilities, the use of a 64-bit time value and a 64-bit secret component in a destructive algorithm were responsible choices to protect authentication over non-promiscuous channels.

If the tokencodes are presented over a medium that can be monitored by an attacker or are viewed on a temporarily borrowed token device, the SecurID implementation is left vulnerable if an attack exists to determine the secret component by viewing previously generated or successive tokencodes. Additionally, with the advent of soft-tokens existing on popular operating systems and not dependent on specific hardware, the level of effort needed to retrieve the proprietary algorithm and secret component has been greatly reduced. If the algorithm is reversible or the initial components to the algorithm are deducible, the risk of cloned cards or prediction of future tokencodes is very real.

The protective measures become simple: ensure the integrity and handling of hardware and software token devices, authenticate through encrypted communications, and, as recommended in [3], ensure that the back-end communications with the application server to the ACE authentication server [4] are not implemented over public links.

With the above stated, the rest of this document will examine areas of concern within the algorithm.

2 Algorithm Concerns

The tokencodes generated by the algorithm are derived from two internal values: a 64-bit time value and a 64-bit secret. The result of the algorithm is then run through a final convert routine which further obfuscates the true algorithm output and produces a value suitable for display on a hardware- or software-based token (Figure 1). The value output by the convert routine is 64-bits in length and is split into multiple tokencodes depending on the token device and display interval.

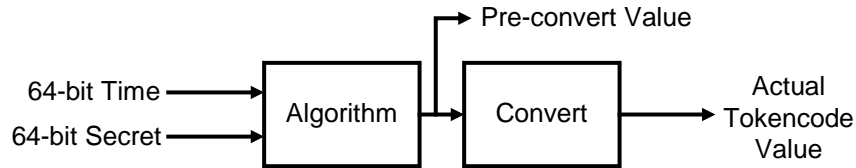


Figure 1: High-level process of tokencode generation.

2.1 Time

The variable input into the SecurID algorithm representing the current time is a 64-bit value. However, examination shows that this 64-bit value is generated from a 32-bit representation of the current time (GMT) in seconds since midnight on 01/01/86² as shown in Figure 2.

```

INT64 time64;
INT32 time;
UCHAR byte;

time = gettimeofday(); // Seconds since 01/01/86, 00:00

// Round down time
time = time / 30;
time = time / 4;
time = time * 4;

// Expand time into 64-bits, duplicate least significant byte
byte = time | 0xFF;
time = time << 8;
time = time | byte;
time64 = time;
time64 = time64 << 32;
time64 = time64 | time;
  
```

Figure 2: Pseudo-code for the 64-bit time computation routine.

For example, should the number of seconds from the SecurID epoch be 0x1C39B862, the time value input to the SecurID algorithm would be 0xF0DB7878F0DB7878. This is derived by rounding the initial number of seconds from the epoch to achieve a value of 0x00F0DB78. From this, the value is shifted left by 8 bits and the least significant byte is represented

²Security Dynamics, the creators of the SecurID card, began operations in 1986.

twice. Thus, it is apparent that only 24 bits are represented from the original 32-bit value representing seconds from the epoch.

Within the rounding function, the value is left shifted twice, which is equivalent to a multiplication by 4. This guarantees, through the associative property of multiplication, that the resultant value will always be even. Hence, the least significant two bits will always be 00 and the only possible values for the least significant nibble are 0x0, 0x4, 0x8, and 0xC. This multiplication removes two more bits of entropy from the 24-bit time value leaving 2^{22} or 4,194,304 total possible time values. It should be pointed out that the possible time values can be further reduced if we assume the attacker has some prior knowledge of the approximate time the tokencode was generated.

The result of this reduced time space is that it is possible to generate a complete tokencode “cycle” by incrementing through all possible values of time for a single secret. The complete cycle contains 4,194,304 total possible 64-bit outputs from the SecurID algorithm. The period of the cycle was designed to be longer than the lifetime of the token device, if the device is run in real-time (in the case where an 8-digit tokencode length is used with a 60-second display interval, it would take ≈ 16 years to cycle through all possible tokencode values). However, using a modified soft-token device, time can be incremented at a much faster rate, thus producing the entire tokencode cycle in a matter of minutes.

2.2 Secret

RSA has not divulged how they generate the initial 64-bit secrets that are used as one of the inputs into their algorithm. As these values are pre-ordained, one can assume that even if the generation mechanism was predictable, it can be changed transparently to a more random generation scheme. However, even with strong pseudo-random or random number generation of the secret, the limited number of possible time values makes this point irrelevant in certain cases.

Should the secret component be compromised from the physical hardware token or application executing the soft-token, all possible output values can be cycled through and recorded by using the proprietary SecurID algorithm as in [1]. This could be done without tampering with the legitimate token device, so the attack would not be immediately detected.

Methods of determining the 64-bit secret from samplings of the token space will be looked at in the more thorough analysis paper.

2.3 Convert

The tokencode value to be entered by the user is represented in decimal ranging from 6- to 8-digits in length. This decimal value is derived from a 64-bit hexadecimal value, called the “pre-convert value”, using a simple transform (Figure 3). This function was designed to map hexadecimal to decimal value in a non-obvious manner and not intended to provide additional security to the SecurID algorithm. The fact that it provides a manner of obfuscation of the pre-convert value is a by-product.

```
For each nibble in the hexadecimal pre-convert value
{
  if (nibble > 9)
  {
    nibble = nibble - 2;
    nibble = nibble - {0, 2, 4, 6, 8};
    nibble = nibble % 10;
  }
}
```

Figure 3: Pseudo-code of simple transform within the convert routine.

For each nibble in the pre-convert value, if the nibble is greater than 9, the transform is used. Otherwise, if the nibble is less than or equal to 9, the natural number is displayed. From this, the value shown to the user could originate as represented in Table 1. In an ideal design, a hex value from 0xA through 0xF could potentially be mapped to a decimal value from 0 through 9. However, since 0xA, 0xC, and 0xE can only be mapped to 0, 2, 4, 6, or 8 and 0xB, 0xD, and 0xF can only be mapped to 1, 3, 5, 7, or 9, the number of possible pre-convert values is greatly reduced.

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
A	B	A	B	A	B	A	B	A	B
C	D	C	D	C	D	C	D	C	D
E	F	E	F	E	F	E	F	E	F

Table 1: Displayed tokencode nibble and its possible pre-convert values

Given $tlen$ as the length of the tokencode, for any given tokencode value the convert routine only provides 4^{tlen} possible pre-convert values, which can then be analyzed to determine particular bit configurations of the actual secret. The ideal design would result in 7^{tlen} possible pre-convert values. For example, a typical 6-digit tokencode has 4^6 or 4096 possible pre-convert values that could have generated it. The ideal case would give 7^6 or 117,649 possible pre-convert values.

For each value of the actual displayed tokencode, there is a 62.5% probability that the number is the natural number (e.g., a 0 displayed has a pre-convert value of 0), and a 12.5% probability that the number is either {A, C, E} or {B, D, F}. Statistical detail is deferred to the more thorough paper.

2.4 Collisions in Tokencode Cycle

Analyzing tokencode cycles for a small sample of pseudo-randomly generated 64-bit secrets yields interesting results. Out of the 8,388,608 possible tokencodes (2 tokencodes produced for each value of time), ≈ 8 million of those tokencodes are unique and occur only once per cycle. The remaining $\approx 300,000$ are repeating or “colliding” values.

$$Collision\ Percentage = \left(1 - \frac{unique\ tokens}{total\ possible\ tokens}\right) \times 100$$

Hence, 4% of the tokencode cycle consists of repeating values. We believe this is partly due to the convert routine.

3 Conclusions

The concerns mentioned in this brief hope to motivate further public assessment of the current SecurID algorithm. Do they negate the usefulness of an infrastructure utilizing this technology? No. However, it does point to the possibility that companies might be assuming more risk than they need to. Hopefully, the detailed concerns provide an opportunity for companies to evaluate how they have deployed this product.

By encrypting the communications, limiting access to back-end communications, and ensuring the integrity and whereabouts of the token generator, the risks of promiscuous viewing of the user authentication and tokencodes and potential retrieval of the secret component are minimized greatly. SSH, DESTelnet, SSL, and other encryption mechanisms can be deployed to help minimize these risks. IPSec, separate back-end management networks, and

other means can be implemented to protect the back-end authentication that occurs between the application server and the ACE/Server.

Users are encouraged to vigilantly protect their hardware token cards or software token key files and not to loan their devices to other people. In addition, extra caution must be taken when utilizing software-based tokens to verify that the host system is not easily compromised.

We are planning for the release of a more thorough analysis of RSA's SecurID algorithm in the near future, which will further detail the statistical irregularities described in this paper. We plan to explore methods of determining the secret component by viewing previously generated token-codes. Additionally, we will examine exhaustive attack scenarios and solutions based upon hypothesized deployment scenarios.

References

- [1] I. C. Wiener, *Sample SecurID Token Emulator with Token Secret Import*, BugTraq posting, December 21, 2000, <http://www.securityfocus.com/archive/1/152525>.
- [2] RSA Security, SecurID Authenticators Web Page, <http://www.rsasecurity.com/products/secuid/datasheets/dsauthenticators.html>.
- [3] PeiterZ, *Weaknesses in SecurID*, <http://www.nai.com/products/security/advisory/papers/secuid.ps>.
- [4] RSA Security, ACE/Server Web Page, <http://www.rsasecurity.com/products/secuid/rsaaceserver.html>.