Weaknesses in SecurID

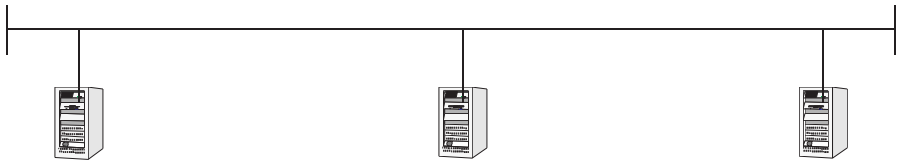
by PeiterZ@silence.secnet.com

This paper was composed for Secure Networks Inc. as research for a commercial network auditing tool with a working name of NAS. For more information on NAS mail nas@silence.secnet.com. This paper has graciously been donated to the public domain. You may redistribute this document as long as proper credit is maintained.

Race attack based upon fixed length response

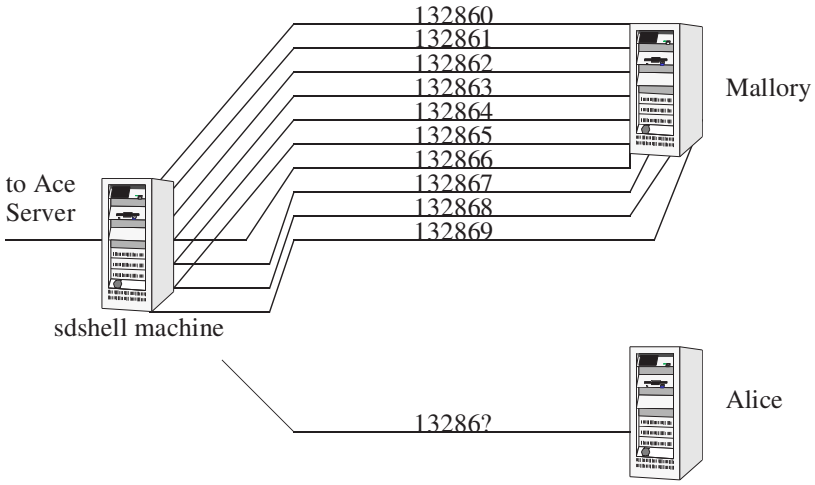
The SecurID token card is set so that it always returns a fixed length string. This string can apparently be four, five or six digits in length. By watching any person login to a system and use the SecurID card (from this point on referred to as simply 'the card'), the attacker knows what the fixed length is for this particular server. The standard that has been witnessed is six digits. These digits range from 000000 to 999999, there is no hex, nor is there any use of alpha characters in the response. Unfortunately, though you might have a total of one million possible numbers to guess, you are limited to 10 possible digits for the last character. Since it is known what position contains the last character the attack is quite simple.

The attacker sits somewhere on the wire between the user and the machine that is running the sdshell program (we do not term this machine 'the server' as Security Dynamics lists this as a client which talks to the ACE server to find out if the hashcode that was entered was valid or not). Note that the attacker does not have to actually be on the physical wire, as is shown below. Any place that an attacker is able to promiscuously monitor the transaction taking place will do for this attack. This is troublesome as one of the primary goals of the card is to prevent the stealing of the password via eavesdropping on the connection.



In the above situation the interloper watches the initial session establishment of a connection from the user machine to the sdshell machine. For this example we will refer to

the user as Alice and the attacker as Mallory. As soon as Mallory sees that Alice is logging in he opens up 10 new connections to the sdshell machine. On each of these connections he simply mirrors in every character that Alice types in her legitimate connection. As Alice enters each character of her username, Mallory sends over the same characters. To the sdshell machine it appears that Alice is logging in eleven (11) times. Given that Alice is a human, coupled with the factors that 1) there is a good possibility that she cannot set her terminal program to LINE BUFFERED mode, 2) numerical keypad entry is usually slower than standard typed responses for most people, and 3) since most of the cards are actual physical tokens (i.e. not something that one would normally be able to cut and past the result from - as would be more feasible in a software based tool) - Mallory has a good chance of winning the race by getting in the correct response on one of his attempts before Alice can enter her last digit and press return.



There are a number of ways that Alice could help to protect herself in this situation.

- Set the terminal / telnet client she is using into Line Buffered mode.

  Most unix terminals will support this - check the appropriate man page on your system.

  Unfortunately many windows and Macintosh clients do not have the option to be set into this mode.
- Type the response from the token card into a local window. *Make sure this is a local window and not a remote session that travels over the network!* From this local window copy the response into a buffer (also called a clipboard on some OS's) including the newline. Once this is completed, paste the entire contents into the session you are responding to.
- Do not allow yourself to become sidetracked while entering your hash-code. The more time you give the attacker before you complete your transaction, the more time the attacker has to use what he/she has already seen to reduce the number of possibly valid responses.

The most recent release of the ACE/Server from SDTI has mechanisms to attempt to prevent this sort of attack. Unfortunately they do not properly protect against the attack and actually open themselves up to a denial of service attack as well.

Denial of Service based upon 's attempt to protect against the fixed length response attack.

The method that is employed to prevent the above attack works as follows: Alice opens up a session to the sdshell machine and Mallory opens up ten (10) other connections to the same machine in order to mirror Alice's keystrokes and race the last digit. The sdshell machine will send its query to the ACE server to see if the user is valid or not after the user has finished entering in their entire hash response. If the ACE server sees more than one request for validation within 2 seconds (default) for the same user then it will disallow both attempts - even if one of them is valid.

Obviously, the denial of service problem here is that Mallory can prevent Alice from ever successfully logging in. All Mallory has to do is set up a program that monitors the connection that Alice is opening and mirror it verbatim. Even though the two connections that are made are identical in their input and both contain the correct hash value from the card, the ACE server will see more than one attempt for validation within its allotted time frame and deny both sessions. There is no error message or information that is presented to either user that alerts them as to why they were denied. The user gets the same response that they would if they had entered an invalid hash code.

What is even more troublesome is the fact that this fix does not prevent the race attack mentioned earlier.

Continued validity of the fixed length response attack with the server fix from SDTI to disallow multiple sign-ons within an allotted timeframe

Since the actual query from the sdshell machine to the ACE server for the validation of a user and their hashcode only happens when the user has entered their entire hashcode and pressed ENTER, the fix from SDTI does not preclude the aforementioned race attack. The window of opportunity is reduced for the attacker but it not removed entirely.

In this situation, Mallory can only send in one packet and has a one in ten chance of choosing the correct last digit. Should he guess correctly, he also needs Alice to have a delay of more than two (2) seconds (or whatever the value on the Ace server is set to) between the time she types the second to last digit and when she finishes entering her entire hashcode. While this might seem a somewhat rare situation to find, it is not that uncommon. There is a sizeable latency apparent in the human interaction of reading the last digit off of the card and entering it into the terminal window. This assumes that there is no form of external interference that might be introduced and thus increase the odds of Alice taking more than two seconds to complete her transaction. Mallory mirrors Alice's connection with a single connection of his own and when Alice enters the second to the last digit Mallory sends across his last digit. Assuming Mallory is lucky and there is a 2 second latency in the amount of time that it takes Alice to enter the last digit and press ENTER, he has a one in ten chance of succeeding. There are various social tacts that Mallory could take such as calling Alice on the phone when he sees her enter the second to last digit in an attempt. Other variants are left to the readers imagination. However, we will shortly show two technical methods Mallory can use to greatly increase his chance of creating the appropriate sized delay that he needs.

One method that Mallory can use to increase his chance of creating the time window is to wait for the appropriate moment (he sees the second to last digit go by) and launch a denial of service attack against Alice's machine. This could be in the form of ping floods, udp storms against localhost, resetting the machines netmask via icmp, RIP attacks, etc. etc.

A more common variant of this attack, which is almost always successful, is to simply reset Alice's connection. Mallory does this by watching the packets going back and forth between Alice and the sdshell machine. As soon as he sees the second to last digit go by all he needs to do is send in the correct forged packet to Alice's machine.

From the sdshell machine to Alice's machine	From Alice's machine to the sdshell machine
	SYN 600
SYN 200 ACK 601	
	ACK 201 SEQ 601
	ACK 201 SEQ 602 #1
ACK 603 SEQ 201	
	ACK 202 SEQ 603 #2
ACK 604 SEQ 202	

ETC. ETC.

	ACK XXX SEQ YYY #n-1
ACK YYY+1 SEQ XXX	
RST YYY+1 SEQ XXX+1	

(this last packet is from Mallory's machine to Alice's machine with a faked address of the sdshell machine)

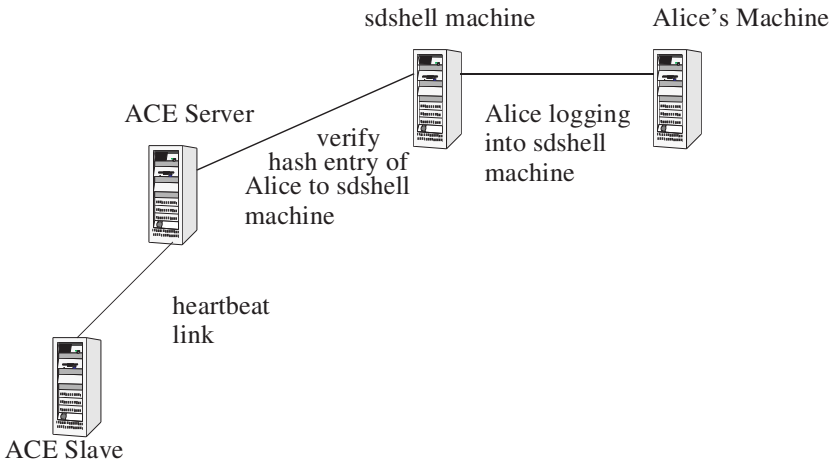
In the above scenario, Alice's connection quietly closes upon receipt of the RST packet that Mallory forged. Mallory now has plenty of time to enter in the digits he saw so far and take a guess at the last number (remember that if the algorithm generating the hash responses is random Mallory does not have to watch to many connections and do this before the guess he makes for the last digit eventually shows up).

Of course, if Mallory is able to RST connections in such a fashion it is almost easier for him to simply allow the connection to complete and then take the whole session over. There are several tools out in the hacker community which make this not only possible but trivial for even people with no knowledge of what is happening at the network and transport layers. This is possible since there is no encryption of the session being performed here. The only thing being provided here is some amount of user authentication at the beginning of the session only. Not only is everything that is done from this point on visible to the attacker (i.e. don't list the private databases during one of these sessions) but it is possible for the attacker to let Alice validate and then *become* Alice as far as the SecurID mechanism is concerned.

Though SDTI does not claim to prevent or protect against this sort of attack it is possible for them to at least help thwart it to some extent. If they were to ask for the hashcode every X number of minutes the attacker would be thwarted after being on the machine for that long. This would be too problematic for many scenarios but could be an option. Keep in mind that even a moderate hacker can get a lot done in a very small amount of time.

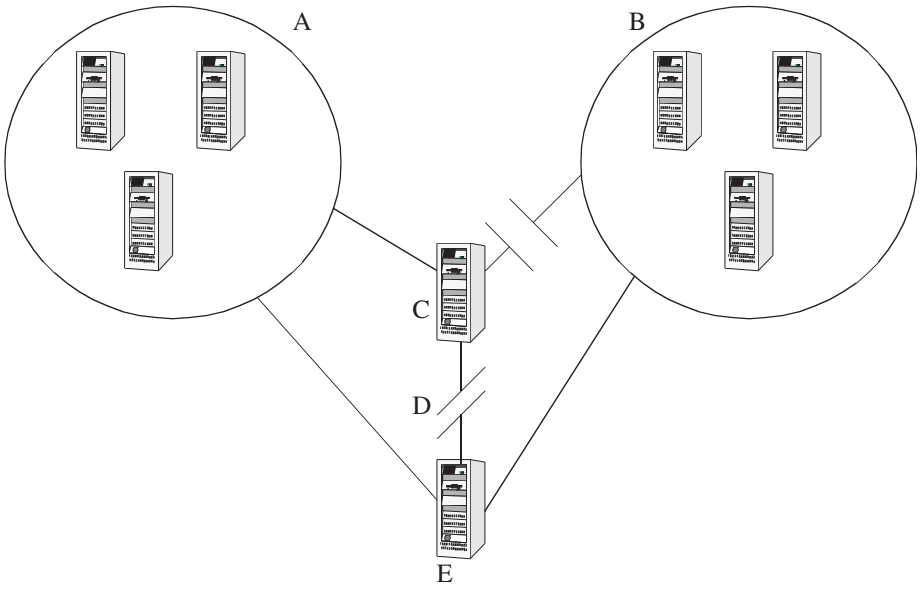
Server - Slave separation and replay attacks

An interesting hole is opened up via the nature of the ACE Server and its slave machine. In a normal situation consisting of a master and slave server the following is the environment we will be dealing with.



What you see in the above picture is user Alice logging into the sdshell machine. When she has entered her username and hashcode the sdshell sends a packet over to the ACE server (more on this problem later). The ACE server determines if that is indeed the correct hashcode for Alice and responds to the sdshell machine with either a '*OK*' or '*Not OK*' packet. The ACE Slave is used in case of problems with the ACE Server. In the event that the Server and Slave drop the heartbeat between each other it is assumed that either something suspicious is happening or that the other machine has simply gone down. Both machines (should they still be up and active) lock their files, so as not to allow any new accounts to be created or existing accounts to be modified. The machine that is still up will start to field the requests from the sdshell machine so that services are not lost to the clients.

The problem arises in the ability to break the heartbeat link between the Server and Slave while having both continue to field responses. By severing the heartbeat, done through any number of denial of service attacks ranging from resetting the netmask of the machine(s) over the network to physically altering the link, it becomes possible to have the master and slave validate the same hash code from the card for different clients. Suppose the network looks like the following:



Key:
A) group of sdshell machines that communicate to the ACE server for validation of hashcodes
B) segregated group of sdshell machines that communicate to the ACE server for validation of hashcodes.
C) the ACE server.
D) severed heartbeat link between the ACE Server and the slave machine.
E) the slave to the ACE server.

Now, assuming this scenario has been created, which is not terribly difficult, both the Server and the Slave are validating hashcodes for separate sdshell machines. Mallory need only sniff a connection to one of the machines in group A and he has a nice time window, until the hashcode is timed out, to go to a machine in group B that the victim has an account on and log in using the same hashcode. There is no contention with successive sign-ons as each is being handled by a unique machine for authentication.

There is an easy fix for this from the standpoint of the ACE machines. As soon as the heartbeat is lost between master and slave, both machines should go into next-token-mode with one machine defaulting to next-token-mode plus one. What Mallory would see, were this enabled, is a prompt for the first hash code which would work on both machines.Next would be a prompt for the next token. Since the Slave would, in this description, be expecting the token after the immediate next-token these would be different responses and thus Mallory would not be able to “mirror” his way in.

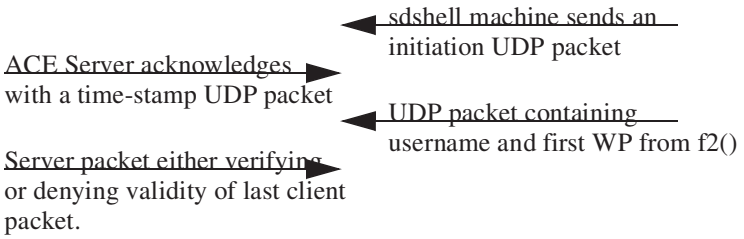
```
if (!heartbeat){
  get_token();
  if (I_am_the_master)
    get_next_token();
```

```
    else{ /* we must be the slave machine */
      sleep(30); /* done to wait until the card token has changed to next-token++ */
      get_next_token();
     }
   }
```

Security Dynamics has been made aware of this problem and has even agreed that this would be a viable fix. However, they still, as of the date of this paper, have not come out with a fix.

Vulnerabilities in the Communications Between the Ace Server and the Client Machine.

When a user connects to the client machine and enters their username and hashcode some interesting communications occur behind the scenes. Upon the initial connection the client machine, which we will refer to as the sdshell machine, sends a UDP packet to the ACE Server. The ACE server acknowledges this packet with a *time packet*. There is no more communication until the user enters their entire hashcode and enters a carriage return. When this event occurs the sdshell machine sends off a packet that contains the following information: username and the first 64 bit WP (more on this in a bit). The ACE server receives this packet and upon decrypting it determines if the request is valid or invalid. If the packet the ACE server received matches what it expected then a response packet equivalent to 'let 'em in' is sent back , encrypted with the second 64 bit WP as the key. If the packet does not match what was expected then the server sends the sdshell machine a response packet equivalent to 'don't allow this attempt'. Before we go into more of the details of how this works and some of the problems with it, let us take a more graphical look at the chain of events.



First, there is one flaw that can be pointed out even from this minimal amount of information. The use of a stateless protocol, UDP in this case, for a mission critical communication goes against every secure coding practice I am aware of. It would have been trivial to have done this communication via a statefull protocol, such as TCP. The coding is only slightly more difficult, depending upon how deep you need to go in the communications process. In an informal discussion with one of SDTI's engineers he concurred that the use of UDP was not a good choice for this communication stream in regards to security.

UDP packets are trivial to modify and send out on the network, appearing to come from anywhere you would like them to. This can be useful for attackers who want to flood the ACE servers default port of 755 or for the more ingenious person who is able to figure

out how to send in a bogus packet to the client machine, appearing to be from the server machine, to self-validate himself.

This is a very interesting thought and has sparked a large amount of discussion in various public and private forums. Security Dynamics claims that their hashing algorithm is secure enough to thwart anyone from breaking it. Thus, it is claimed, these types of attacks are not a threat. Unfortunately, the algorithm is proprietary - thus the cryptographic community is not able to perform peer reviews on it to confirm or deny this belief. Given the track record of how poorly proprietary cryptographic systems and even public systems have fared against the crypto-community it is quite likely that this algorithm would prove to be vulnerable in certain areas. Customers should question claims of security through obscurity.

Here is what is happening inside the packet exchange in the above diagram:

The client sends a udp packet to the Ace server's listening port (usually 755). This packet contains an identifier of

> 0x67 - 0x02 - 0x00 - 0x10
>
> followed by padding.

This particular identifier signifies that the server should send back a time packet.

The Ace server then sends back the time packet which is symbolized by the identifier of:

> 0x6c - 0x02 - 0x00 - 0x10
>
> followed by a time data. [note: All SDTI information that I have been given refer to the contents as a time packet, this would also make sense in the order in which it is sent. I have not spent any time looking into the format of the information in this packet as of yet]

After the user has entered their username and token, a packet with the following is sent out to the server:

> 0x65 - 0x02 - 0x00 - 0x10
>
> followed by 4 bytes of padding and then the username of the person attempting to login in plaintext [note: Security Dynamics has claimed that this entire packet is encrypted but it is obviously not the case as the username is plainly visible] followed by padding to the 85th byte of the UDP packet. And finally, 48bytes of data - 64 bits of which supposedly are the first WP from the f2() hash.

The final response from the Ace server does not appear to have a unique ID. It appears to contain entirely encrypted information (most likely with the ID inside). It is worth noting that the UDP 16 bit checksum seems to always be zero. This is the packet that allows or denies access.

This is all fine information but does still not explain some of the problems. For this we need to look at the encryption mechanism used for the data inside these packets. This

information has been gleaned from various communications with SDTI engineers and the small section on their protocol in the SecurID FAQ.

The time packet is said to be DES encrypted with the client machines DES key. The client takes this information and together with the hashcode entered by the use and it's own IP address, creates a 256bit hash string.

```
256bit_result = f2(time, hashcode, IP);
```

The 256 bit string is broken into 4 64 bit chunks, referred to as WP's. The client then sends the username and the first WP to the server. The server f2()s the same data using the hashcode that it believes to be valid for the current time, along with f2()s for the previous and next expected hash code. If the first WP from any of these matches the WP that was received from the client the request was good. Upon receipt of a good request the server then encrypts the *OK packet*, as I refer to it, using the second WP as the DES key.

It is apparent that the f2 function uses too much easily gleanable information. The hashcode could be known, the time can be gotten from the system in question in any number of ways, and the IP address is a constant. If Mallory is able to figure out the f2 function then the following attack might be possible:

Mallory connects to a client machine xxx.xxx.xxx.xxx and logs in using the username Alice. He then enters the hashcode of 111111. Mallory knows the username that will be in the second UDP packet, the hashcode that will be used in the f2 function, the IP address that will be used in the f2 function, and we will suppose that he can determine the time. If Mallory knows what an *OK packet* should contain and is able to compute the 4 WP's before he logs in, he can encrypt the response packet with the second WP string and use this packet to authenticate himself. It would have been more difficult had the underlying session been TCP but again, not impossible. Conversely, it might be possible for Mallory to pose denial of service threats by constantly sending back *Not OK packets* before the real server can respond.

The above scenario is based upon the information that has been presented by SDTI largely in the form of their FAQ. Conflicting stories have come out from various phone conversations with SDTI employees. It is not known whether all of the semantics are absolutely correct in this example but it is quite probable that some variation of the attack is possible. This is one of the problems with keeping security related functions proprietary. The onus should be on the vendor to prove that it is secure by publicly proving the validity of its statements.

In other papers and talks I have mentioned that SecurID would still be a viable option given that it was used inside of some form of encrypted telnet. However, since the communications between the client machine and the server are done out-of-band; if the above attack is performed, even an encrypted telnet session would not help.

A SecurID sdshell bug in versions pre 2.2 and 1.3.1

It appears that in SecurID versions prior to 2.2 and 1.3.1 (which has not been released as of the Aug. 16 1996) it is possible for the sdshell to become confused. Upon logging in and executing the sdshell, there are situations that make the sdshell believe that the person is su'ing and not logging in. In particular it seems to believe that the user is root and

is su'ing to another entity. The following packet exchange occurs between the client and the server. This helps to illustrate how important the underlying communications between the server and the client can be.

Although everything is set up 'properly' on the server and the user is given a shell of sdshell on the client machine, a prompting and acceptance of the users unix password and not the token hash happens.

The client sends the standard packet requesting a time packet from the server

0x67 - 0x02 - 0x00 - 0x10

The server responds with the time packet:

0x6c - 0x02 - 0x00 - 0x10

The client then sends the info packet with an ID specifying that this is for an su and that it needs the persons shell to exec.

0x66 - 0x02 - 0x00 - 0x10

Followed by 4 bytes of 0x00 padding, the username, padding to byte 86 of the UDP packet, 16 bytes of data, and more padding.

Server responds with following packet:

0x6c - 0x02 - 0x00 - 0x10

Followed by 8 bytes of unknown, the users shell in plaintext, padding to the 86th byte in the UDP packet, and more data.

The client now knows what shell the user is going to and sends another packet asking for, proposedly, authentication.

0x76 - 0x02 - 0x00 -0x10

Followed by 4 bytes of padding, the users name in plaintext, padding to the 86th byte in the UDP packet, 16 bytes of data, and more padding.

The server then responds with its acknowledgment packet, keeping in mind that the UDP checksum is not set.

Conclusion

It seems that SecurID was a valid tool when it first was introduced but is now suffering in security areas as the world has evolved around it. The card does up the level of attacks necessary but does not seem to offer the security that one might expect it to. In the world of computer and network security some of the biggest vulnerabilities occur when there is a false sense of security. Particular concern is felt for large corporations that rely solely upon this method for 'secure access'. The race attacks based upon fixed length responses, denial of service attacks, separating the master and slave servers and the sdshell bug have all been witnessed. The weakness based upon understanding the f2 hash has not been witnessed to the best of my knowledge. However, if there is a problem there, it is only a matter of time before it is exploited. Reverse engineering the f2 hash should not be difficult for anyone with a copy of the sdshell binary and a common debugger. This paper is meant to inspire customers, consumers, and hackers alike to

question companies when they claim their product is secure and that there are no vulnerabilities in it - even more so when the security is based upon proprietary software algorithms.

Credits:

- LHI Technologies was invaluable with their help in analysis of the hardware token cards. Who knows, maybe there will be a paper on those in the near future.
- Adam Shostack initially pointed me to the SecurID FAQ which showed the easily obtainable information that is used as the arguments to the f2 hash function.
- hobbit@avian.org was great at listening to me rant and pointing out things that I had missed / correcting flawed logic of mine when it popped up [quite frequently]. He is also the one who kept harping on me to ‘just RST the connection after you’ve seen all of the numbers and before the <c/r>!’.