Linux's answer to MS-PPTP

1 message

Peter Gutmann <pgut001@cs.auckland.ac.nz> **Mon, Sep 22, 2003 at 2:28 PM**
To: cryptography@metzdowd.com

A friend of mine recently pointed me at CIPE, a Linux VPN tool that he claimed was widely used but that no-one else I know seems to have heard of. Anyway, I had a quick look at it while I was at his place. It has some problems.

CIPE lead me to another program, vtun, which is even worse. Someone else then told me about another one, tinc, which apparently was just as bad as CIPE and vtun, but has been partially fixed after flaws were pointed out (it still has some problems, see below). The following writeup covers some of the problems, with conclusions drawn at the end. As you'll note from reading this, as I went down the list of VPN software I got less and less interested in writing up lengthy analyses, so CIPE (the first one I looked at) has the most detail.

CIPE
====

The following comments on CIPE apply to the protocol described at http://sites.inka.de/sites/bigred/devel/CIPE-Protocol.txt (the CIPE home page).

Section 2, Packet Encryption:

- CIPE uses a CRC-32 for integrity-protection. The use of a weak checksum under CFB or CBC has been known to be insecure for quite some time, providing no (or at least inadequate) integrity protection for the encrypted data. This first gained widespread attention in 1998 with the Core SDI paper "An attack on CRC-32 integrity checks of encrypted channels using CBC and CFB modes" by Ariel Futoransky, Emiliano Kargieman, and Ariel M. Pacetti of CORE SDI S.A, which lead to the SSHv1 insertion attacks and the more or less complete abandonment of SSHv1. To quote the Core SDI work:

  The use of these algorithms in CBC (Cipher Block Chaining) or CFB (Cipher Feedback 64 bits) modes with the CRC-32 integrity check allows to perform a known plaintext attack (with as few as 16 bytes of known plaintext) that permits the insertion of encrypted packets with any choosen plaintext in the client to server stream that will subvert the integrity checks on the server and decrypt to the given plaintext, thus allowing an attacker to execute arbitrary commands on the server.

  In other words this provides no real integrity protection for data. Although it first gained widespread publicity in the SSHv1 attacks, the fact that this mechanism is insecure and shouldn't be used goes back to (at least) Kerberos v4 dating from the 1980s (the des-cbc-crc mechanism was deprecated in Kerberos v5, the now ten-year-old RFC 1510).

- The padding length is limited to 3 bits, making it unusable with any recent 128-bit block cipher. In particular, AES can't be used. In addition, the inability to pad to more than one (64-bit) cipher block length makes it impossible to disguise message lengths by padding messages to a fixed size (there are further SSHv1 attacks that arose from similar problems there). This weakness is particularly problematic when applied to section 3.

- There is no protection against message insertion or deletion. In particular an attacker can delete or replay any message, and in combination with the weak checksum problem can replay modified messages. Consider for example what would happen if an attacker can replay database transactions where money is involved. This issue is also particularly problematic when applied to section 3.

Recommendation to fix:

This portion of the protocol has a number of flaws, but can be fixed with a more or less complete overhaul of the message format:

- Replace the weak checksum with a MAC like HMAC-SHA1. If size is a concern, use a truncated HMAC (but not to a mere 32 bits, which is unconvincing). IPsec cargo-cult protocol design practice would seem to require a MAC size of at least 96 bits (IPsec truncated the MAC to 96 bits because that makes the AH header a nice size, with a payload length of exactly 128 bits (4 32-bit words); everyone else assumed that the number 96 had some magic significance and copied it into their own designs).

- Replace the padding with standard PKCS #5 padding, allowing both the use of ciphers with block sizes > 64 bits and padding of messages to hide plaintext size.

- Either provide protection against insertion/deletion via message sequence numbers, or make it very explicit in the documentation that CIPE should not be used where insertion/deletion can be a problem, i.e. in situations where the higher-level protocol being tunneled doesn't provide its own mechanism for detecting missing, inserted, or out-of-sequence messages.

A quick fix would be to take the SSL (or SSH) format, strip out the SSL headers and encapsulation, and use what's left (the padding, MAC'ing, etc etc). SSL/SSH also provides message-sequence handling if you want this.

Section 3, Key exchange:

- The lack of integrity protection means that it's possible to modify keys in transit. As an extreme example, if 3DES keys were used it'd be possible to flip the necessary bits to force the use of weak keys, so that both sides would end up sending plaintext. CIPE doesn't appear to use 3DES, but does use IDEA, which also has weak keys. In any case having an attacker able to set key bits in arbitrary ways is never a good thing (they could presumably force the use of an all-zero or otherwise known key, which is only marginally better than sending in plaintext).

- The lack of replay protection means messages containing instructions to switch to old keys can be replayed. In particular, re-use of a compromised key can be forced in this way.

- Since packets are ignored if the checksum is bad, it's possible to no-op out key-change messages (forcing continued use of a compromised key) by flipping a bit or two.

- The lack of ability to mask the content length allows key management packets to be quickly identified by an attacker, and the above attacks applied. For example it looks like keyex packets will always be 24 bytes long, while tunneled TCP packets will never be that short. In any case keyex packets

 are ID'd by the (plaintext) flag that indicates which key type is being
 used.

I'm sure I could find more problems if I thought about it a bit more, but I think that's about enough - see below for more.

Recommendation to fix:

Basically, this part of the protocol is beyond repair. Any active attacker can cause about the same level of havoc that Schneier et al managed for Microsoft's original PPTP implementation. The fix for this is to scrap the key exchange portion completely and replace it with an SSH or SSL management tunnel, which provides the necessary confidentiality, integrity protection, authentication, message insertion/deletion protection, etc etc. Since control messages are rarely sent, there's no performance penalty to using a TCP channel for this portion of the protocol. The alternative would be to more or less reinvent SSH/SSL using CIPE, which doesn't seem like a useful exercise.

(Author contacted, no response).

vtun
====

While googling for more CIPE info, I found another Linux encrypted tunnel package, vtun. Ugh, this makes CIPE look like a paragon of good crypto design in comparison. Although the format is undocumented, from a quick look at the source code it appears that vtun uses Blowfish in ECB mode with no integrity protection, and as for CIPE no protection against message insertion or deletion, replay attacks, etc etc. Eeek! Then there are code fragments like:

```
void encrypt_chal(char *chal, char *pwd)
{
  char * xor_msk = pwd;
  register int i, xor_len = strlen(xor_msk);

  for(i=0; i < VTUN_CHAL_SIZE; i++)
     chal[i] ^= xor_msk[i%xor_len];
}

[...]

void gen_chal(char *buf)
{
  register int i;

  srand(time(NULL));

  for(i=0; i < VTUN_CHAL_SIZE; i++)
     buf[i] = (unsigned int)(255.0 * rand()/RAND_MAX);
}
```

which sort of speak for themselves. Furthermore, the key is just a straight hash of the password (no salt, iterated hashing, or other standard precautions are used). I feel somewhat bad for the author(s) of vtun because it looks like there's been a fair bit of work put into it, but honestly my best recommendation for this in terms of security is to scrap it and start again.

(Some more googling for vtun info found a post by Jerome Etienne from January
2002 pointing out these exact same problems, so the fact that it has horrible
security problems has been known for quite some time. Nothing appears to
have been fixed in the nearly two years since the problems were pointed out).

tinc
====

So I sent a rough set of notes out for comment and someone pointed me to tinc.
tinc uses a completely predictable 32-bit IV in combination with CBC
encryption, which makes the first encrypted block vulnerable, but isn't quite
as bad as the ECB used in vtun (actually it isn't an IV in the true sense, it
prepends a 32-bit sequence number to the encrypted data, but it works the same
way). Packets are protected with a 32-bit (very) truncated HMAC using
encrypt-then-MAC.

tinc's real problem though is the handshake protocol, in which the client and
server exchange random RSA-encrypted strings. That's raw bit strings, there's
no PKCS #1 or OAEP padding, and the server is happy to act as an oracle for
you too. This is a terrible way to use RSA, and usually compromises the key.
There is a significant body of literature on this (too much to cite) going
back to the early 1980s and continuing through to the current day, with
probably the most recent publication in this area being the attack published
at Usenix Security '03 only a few weeks ago (in that case it was a timing
oracle).

Beyond that, the protocol writeup
(http://tinc.nl.linux.org/documentation/tinc_6.html#SEC61) points out that:

 the server sends exactly the same kind of messages over the wire as the
 client

In case the problem isn't obvious yet, note that what's being exchanged is
purely a random bit string, with no binding of client or server roles or
identities into any part of the exchange. Again, there are way too many
references to cite on these issues, although my favourite coverage of a lot of
the things you need to think about is in "Network Security: Private
Communication in a Public World" by Kaufman, Perlman, and Speciner. As an
example, here's a simple attack. The documentation (section 6.3.1) is a bit
vague about the message flow, but assuming I've understood it correctly, the
message flow is:

```
 client                          server
            rsa( random_key ) -->
          random_key( challenge ) -->

      <-- random_key( sha1( challenge ) )
```

Simplifying things a bit so that the entire exchange can be abstracted down to
"challenge" and "response" (with implied RSA decryption, etc etc taking place
as part of that), let's say Mallet wants to mess with Alice and Bob. So
Mallet sends a challenge to Bob (claming to be Alice) and gets back a
response. Mallet gets Bob's encrypted key and challenge back and forwards it
to Alice, who returns a response, which Mallet in turn forwards to Bob, a
classic chess grandmaster attack. Bob now thinks he's talking to Alice, when
in fact Mallet controls one of the two channels.

If the exchange is something like rcp (where all traffic is one-way, you write
the entire file and close the connection), that's all that's needed, Mallet
just sits there sucking down the data that Bob thinks is going to Alice. If
not, Mallet does the same thing to Alice, who thinks she's talking to Bob.
Again, Mallet now has a one-way channel to Alice. Mallet can then splice the
two channels that he controls, so he has a channel in both directions. Again,
depending on the protocol being tunneled, it may be possible for Mallet to use
the write channel he controls to get data sent on the read channel he
controls.

As an extension of the handshake problem, tinc relies heavily on an
administrator at both ends of the link configuring the software identically
for the handling of the handshake phase, replacing the authenticated parameter
negotiation performed by protocols like SSL/TLS, SSH, and IPsec (during the
data transfer phase there are ways of negotiating parameters, I haven't looked
at this too much).

Overall, tinc isn't anywhere near as bad as CIPE or vtun, but like CIPE it
should have the handshake/control channel replaced with an SSH or SSL tunnel,
and the data-transfer portion could be improved as well. The tinc handshake
protocol would also make a nice homework exercise for students in computer
security courses, since it's not trivially broken but does exhibits various
textbook handshake-protocol flaws, violating a great many of the general
principles of secure protocol design:

- Don't use the same key for encryption and authentication.

- Don't have the key chosen entirely by one side (even if it's only Alice
  and Bob on the line rather than Mallet, if one of the two has a poor RNG,
  all communications from them are compromised).

- Provide message freshness guarantees (and, if possible, some form of PFS).

- Bind the identity of the principals to the authentication.

- Don't use the same messages in both directions.

- Don't act as an oracle for an attacker.

As it stands the handshake protocol only narrowly avoids a variety of basic
attacks, making it quite brittle in that the most trivial change (or someone
thinking about possible attacks a bit further :-) would probably kill it.

(Following on from the footnote at the end of the vtun bit, the same Jerome
Etienne who pointed out problems in vtun also picked apart an earlier version
of tinc, which looks to have been at about the same level as CIPE when he
looked at it. A lot of it appears to have been fixed since then.

Authors contacted, their response was that the problems were not that
serious, and that a main goal of tinc was to get the opportunity to
experiment with secure tunneling protocols and mechanisms (other design goals
are at http://tinc.nl.linux.org/goals).

Thoughts
========

Several points arise from this writeup:

- These programs have been around for years (CIPE goes back to 1996 and vtun
  to 1998) and (apparently) have quite sizeable user communities without
  anyone having noticed (or caring, after flaws were pointed out) that they
  have security problems. I only heard of CIPE when a friend of mine
  mentioned it to me in passing, and came across vtun by coincidence when I
  was looking for more info on CIPE. Who knows how many more insecure Linux
  crypto-tunnel products there may be floating around out there.

- It's possible to create insecure "security" products just as readily with
  open-source as with closed-source software. CIPE and vtun must be the OSS
  community's answer to Microsoft's PPTP implementation. What's even worse is
  that some of the flaws were pointed out nearly two years ago, but despite
  the hype about open-source products being quicker with security fixes, some
  of the protocols still haven't been fixed. At least Microsoft eventually
  tries to fix their stuff, given sufficient public embarrassment and the odd
  hundred thousand or so computers being taken out by attackers.

- For all of these VPN apps, the authors state that they were motivated to
  create them as a reaction to the perceived complexity of protocols like SSL,
  SSH, and IPsec. The means of reducing the complexity was to strip out all
  those nasty security features that made the protocols complex (and secure).
  Now if you're Bruce Schneier or Niels Ferguson, you're allowed to reinvent
  SSL ("Practical Cryptography", John Wiley & Sons, 2003). Unfortunately the
  people who created these programs are no Bruce or Niels. The results are
  predictable.

- Whenever someone thinks that they can replace SSL/SSH with something much
  better that they designed this morning over coffee, their computer speakers
  should generate some sort of penis-shaped sound wave and plunge it
  repeatedly into their skulls until they achieve enlightenment. Replacing
  the SSL/SSH data channel is marginally justifiable, although usually just
  running SSL/SSH over UDP would be sufficient. Replacing the SSL/SSH control
  channel is never justifiable - even the WAP guys, with strong non-SSL/SSH
  requirements, simply adapted SSL rather than trying to invent their own
  protocol.

Thanks to Greg Rose for checking a draft copy, and Matt Robinson for inspiring
the sound-wave comment (I wonder how many mail filters that'll get caught
on?).

Peter (sigh).