Global Information Assurance Certification Paper

An Integer Overflow Attack Against SSH Version 1 Attack Detectors

David J. Bianco
March 1, 2001
Assignment version 1.2d

Introduction

On June 11th, 1998, Ariel Futoransky and Emiliano Kargieman of Core SDI, the Argentinean information security research group, published an advisory detailing how it was possible for an attacker to insert arbitrary information into the encrypted data stream between an SSH version 1 client and server. SSH version 2 clients and servers were not vulnerable. Due to architectural issues in the version 1 protocol, the attack could not be prevented without changing code, which would render the new versions of the software incompatible with existing version 1 code. Instead, they chose to pursue a strategy that would allow the clients and servers to detect that such an attack had already taken place and thus allow the program to react. Core SDI released a file called deattack.c which implemented this function. The file was subsequently incorporated into most implementations of SSHv1 clients and servers.

Ironically, it is this attack detection routine which is itself vulnerable to an integer overflow attack. This paper describes the attack, lists software known to be vulnerable and provides technical detail on the attack itself and on prevention.

Description

The suggested fix to the original problem included a new routine, detect_attack(), which defined a 16-bit local variable. This variable was used in conjunction with 32-bit local variables and due to 16-bit integer overflow, it was possible to send long (length > 2^{16} bits) input packets to the server or the client which would cause this variable to be set to an effective value of 0. This led to unintended side effects in the code, such as an array table overflow, which would allow an attacker to modify arbitrary addresses within the address space of the program. This attack can be executed without having to successfully authenticate to the target system. ***Table 1*** shows several popular SSHv1 implementations and their vulnerability state. Even if not explicitly listed, all versions of SSHv1 that use the attack detector should be considered vulnerable.

Program/Vendor	Version	Notes
OpenSSH	All versions prior to 2.3.0 are vulnerable. 2.3.0 and later are not vulnerable.	
SSH.com	Versions 1.2.24 through 1.2.31 are vulnerable. Versions prior to 1.2.24 did not include the attack	SSH.com recommends that users of v1 software upgrade to SSH2. Nevertheless, a patch has

	detector, and thus are not vulnerable. Version 2.x is not vulnerable	been applied to their source tree and future versions of the v1 product will not be vulnerable.
F-Secure SSH	Version 1.3 is vulnerable	
AppGate	See notes	The server is not vulnerable in the default configuration because v1 support is disabled. If v1 support has been enabled, contact the vendor for a fix.
TTSSH	Not vulnerable	
LSH	Not vulnerable	LSH does not support the v1 protocol.
JavaSSH	Not vulnerable	There is no attack detection mechanism in the code.
OSSH (by Bjoern Groenvall)	Version 1.5.7 and below are vulnerable. Version 1.5.8 is not vulnerable.	
Cisco SSH	Not vulnerable	They have implemented their own code, and thus do not use the vulnerable attack detector.
Van Dyke Technologies SecureCRT	Unknown.	

Table 1: SSHv1 Implementations

Detail

The function signature for detect_attack() looks like the following[1]:

```
/*
   detect_attack
   Detects a crc32 compensation attack on a packet
 */int
detect_attack(unsigned char *buf, word32 len, unsigned char *IV)
{
    static word16  *h = (word16 *) NULL;
    static word16   n = HASH_MINSIZE / HASH_ENTRYSIZE;
    register word32 i, j;
    word32          l;
```

The line in ***bold italics*** shows the first troublesome variable, *n*. Although *n* is declared as a 16-bit integer, it is later used in conjunction with 32-bit values, which leads the root of the problem.

```
for (l = n; l < HASH_FACTOR(len / SSH_BLOCKSIZE); l = l << 2);
        if (h == NULL) {
        debug("Installing crc compensation attack detector.");
        n = l;
```

```
        h = (u_int16_t *) xmalloc(n * HASH_ENTRYSIZE);
}
```

As you can see in the code sample[1] above, the loop control variable *l* is left-shifted at the end of each iteration. If the length of the buffer in the incoming packet (*len*, in this example) is sufficiently large, *l* will eventually grow to the value of 65536, which is just one bit larger than can be stored in the the 16-bit value *n*, as is shown inside the loop. This will cause the integer overflow, causing *n* to become 0. In the next line, a call to xmalloc() with an parameter effective parameter of 0 (after all, 0 * HASH_ENTRYSIZE always equals 0), will cause the variable *h* to become a valid pointer to a zero-length object within the program's namespace.

Now the real fun begins.

Consider the following code[3]:

```
for (c = buf, j = 0; c < (buf + len); c += SSH_BLOCKSIZE, j++)
{
  for (i = HASH(c) & (n - 1); h[i] != HASH_UNUSED;
       i = (i + 1) & (n - 1))
  {
    if (h[i] == HASH_IV)
    {
      if (!CMP(c, IV))
      {
        if (check_crc(c, buf, len, IV))
          return (DEATTACK_DETECTED);
        else
          break;
      }
    } else if (!CMP(c, buf + h[i] * SSH_BLOCKSIZE))
    {
      if (check_crc(c, buf, len, IV))
        return (DEATTACK_DETECTED);
      else
        break;
    }
  }
 h[i] = j;
}
```

The outer loop simply breaks up the one large packet into chunks of length *SSH_BLOCKSIZE*. By default, this means that it processes the buffer 8 bytes at a time. You'll notice fairly quickly that the second loop (the one using *i* as the loop counter) contains a logical AND of the value of *HASH(c)* and *n*. In this case, *c* is the variable containing the data in the packet that we're checking. Since *n* is set to an effective value of 0, *(n – 1)* comes out to be –1, which of course is represented by the bytes 0xffff (all binary 1). If you AND a value with 0xffff, you get the original value back, thus this piece of code is effectively equal to the following:

```
for (i = HASH(c); h[i] != HASH_UNUSED; i = (i + 1))
```

Unfortunately, the HASH() function in this case doesn't really do anything special. It simply fetches gets the first 4 individual bytes from this 8 byte chunk and treats them as though they were one long 32-bit integer, which it returns.

Since this "hash" is used as the array index, by crafting your own packet, you can control the contents of the first 4 bytes of each chunk, thus, you can control the array index. Remember that *h* is always a valid pointer to somewhere within the memory address space of the running program. You don't necessarily know where it is, though, which makes things tricky. Let's say that the 4 bytes you choose are all 0xff. The array indexes *h[i]*, which comes out to be *h[0xffffffff]* or the address 0xffffffff bytes offset from the address pointed to by *h*. In most cases, you're pretty much stuck with guessing where *h* is really pointing to, though, which is one of the things that makes this a difficult hole to exploit. More on this a little later on.

Ok, so now you've got an offset somewhere in memory. What next? How can you assign a value of your own choosing to the address? The answer is simple. Recall the second to last line of the above sample, *h[i] = j*. In this case, *j* is a simple loop counter variable. If you want to place a value of, say, 10, into your own custom *h[i]*, you craft your packet such that the 10th chunk contains the offset you want to write to. You're limited to writing only loop counter's value to your chosen memory address, but since the maximum packet size in SSHv1 is 256K, you can pretty much take your pick of values. By sending multiple packets, you can see how easily an attacker could build a chunk of malicious code in the program's address space.

It's worth noting that although *h[i]* is offset from *h* by *i* bytes, if *i* is sufficiently large, it could wrap around and start pointing to the beginning of the address space, so it's possible to point anywhere in the program. If the attacker doesn't accidentally crash the server, they could modify variables or other important values rather than simply insert their own code.

Many of those who have issued Internet security bulletins about this attack have noted that there is an additional step required to successfully inject your own custom packets into the SSH transmission. The receiving SSH client or server process will expect that your packet has already been encrypted with the negotiated symmetric session key. Thus, sending your crafted packet in the clear would cause it to contain the wrong values after it is "decrypted" and sent through detect_attack(). This is easily solved by negotiating the beginning of an SSH session, which establishes the session key. With this, it is straightforward (in fact, almost trivial) to perform the proper encryption with your favorite plaintext bytes.

Although this exploit is an extremely difficult one to exploit, code has already been posted [3] to the Internet that can successfully attack OpenSSH 2.1.1 if the 'emptypasswords' option is set on the server to allow NULL passwords to be legal. This code works by writing to a predetermined offset for the variable that holds the *pw->pwent* data for the account the session is trying to log into. The attacker can

presumably find this offset by compiling the same SSH server version on his own machine of the same type he is attacking (with the same compiler and compilation settings, of course). In this case, the exploit simply makes the first byte in this string a NULL, so the attacker can enter a NULL password on the client and log in to any account, including root.

Fixes & Recommendations

The best fix most vendors recommend, of course, is to upgrade to SSHv2 compliant software if possible, then disable all v1 support. In cases where this is not possible, contact your vendor immediately and obtain a patch for your software. If you have a source distribution, Core SDI has already released the fix, which is simply to change *n*'s data type to a 32-bit integer, as shown in the following patch file for SSH.com's SSH v1.2.31[1].

```
--------------------- begin deattack patch ------------------
This is the patch for ssh-1.2.31 package.
Using the patch:
Copy the ssh-1.2.31.tar.gz package and the ssh-1.2.31-
deattack.patch in a directory.
Decompress the ssh-1.2.31.tar.gz package:
 tar xzvf ssh-1.2.31.tar.gz
Apply the patch:
 patch < ssh-1.2.31-deattack.patch
Compile the ssh package.
--- ssh-1.2.31/deattack.c-old Wed Feb  7 19:45:16 2001
+++ ssh-1.2.31/deattack.c Wed Feb  7 19:54:11 2001
@@ -79,7 +79,7 @@
 detect_attack(unsigned char *buf, word32 len, unsigned char IV)
 {
   static word16  *h = (word16 *) NULL;
-  static word16   n = HASH_MINSIZE / HASH_ENTRYSIZE;
+  static word32   n = HASH_MINSIZE / HASH_ENTRYSIZE;
   register word32 i, j;
   word32          l;
   register unsigned char *c;
--------------------- end deattack patch ------------------
```

References

1. Core SDI, "SSH1 CRC-32 compensation attack detector vulnerability", 8 February 2001, URL: http://www.core-sdi.com/advisories/ssh1_deattack.htm (1 March 2001)

2. Core SDI, "SSH INSERTION ATTACK", 11 June 1998, URL: http://www.core-sdi.com/advisories/ssh-advisory.htm (1 March 2001)

3. Starzetz, Paul, "ssh1.crc32.txt", 21 February 2001, URL: http://packetstorm.securify.com/0102-exploits/ssh1.crc32.txt (1 March 2001)

4. Unknown, “OpenSSH FAQ”, 26 February 2001, URL: http://www.openssh.com/faq.html (1 March 2001)

5. Zalewski, Michal, “Remote vulnerability in SSH daemon crc32 compensation attack detector”, 8 February 2001, URL: http://razor.bindview.com/publish/advisories/adv_ssh1crc.html (1 March 2001)