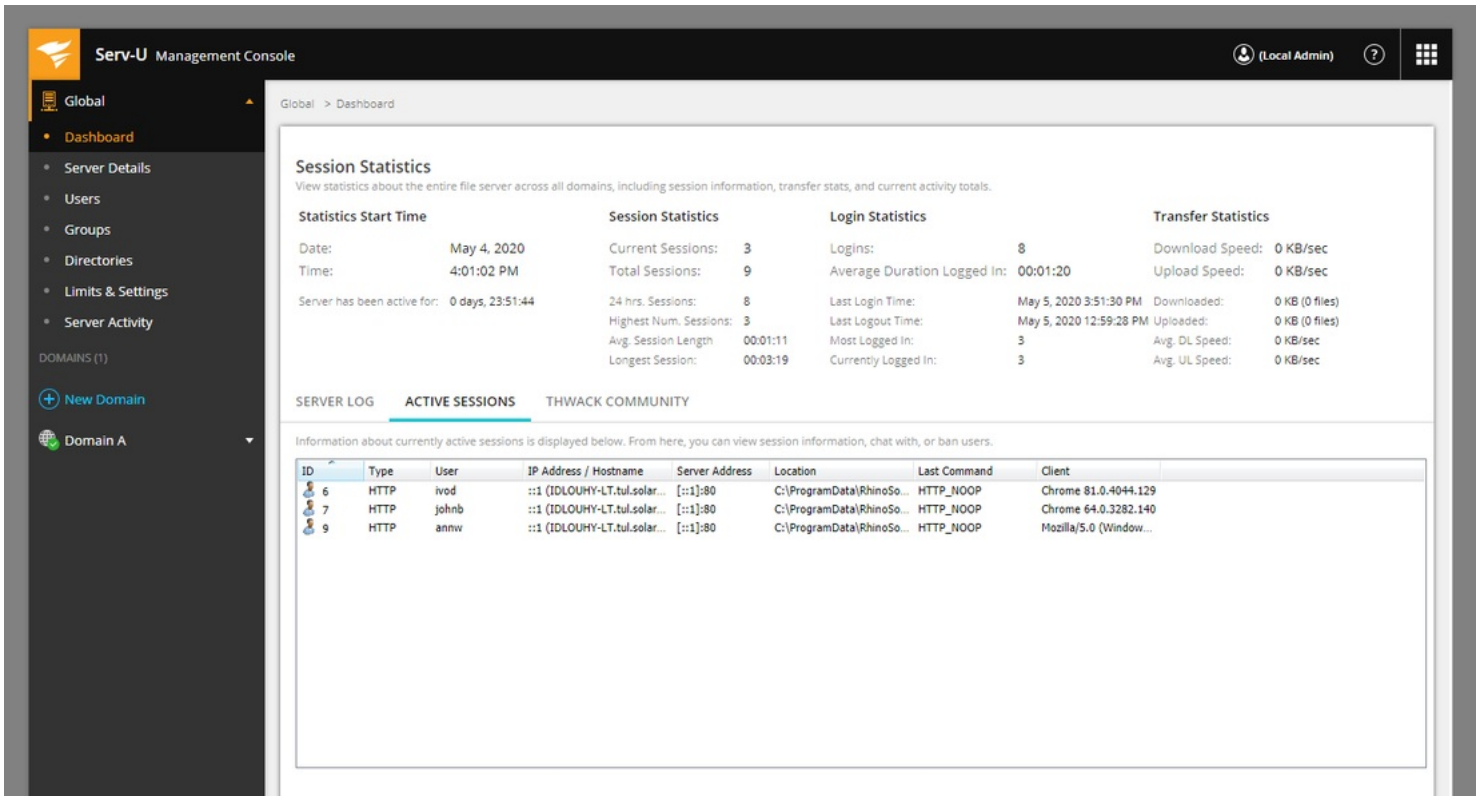
Analysis of CVE-2021-35211 (Part 1)

17 September 2021

Introduction

On around 13 Jul, I chanced upon this article warning users of Solarwinds Serv-U against a pre-auth SSH RCE bug being exploited in the wild. This was pretty interesting to me, as I didn't think SSH RCE was still possible in a year like 2021. What followed was a 2 month long on and off exploration of the Serv-U SSH codebase. In the end, the original team at Microsoft released a (more authoritative) writeup a few days before I figured out the bug. Nonetheless, the following two-part series would hopefully present a different perspective on the vulnerability analysis process. Part 1, or this post, will go through my process behind identifying and triggering the vulnerability, while part 2 will demonstrate how I made use of the vulnerability to gain RCE. Do note that all this research were done on the Windows version of Serv-U.

What is Serv-U?



Serv-U

Solarwinds Serv-U is a file-sharing server for both Windows and Linux. One of its features is to allow file transfers via SFTP, a file transfer protocol built on top of the SSH protocol. CVE-2021-35211 is a memory corruption vulnerability in the SFTP component of Serv-U which can take place even before a successful SSH login, and affects Serv-U 15.2.3 HotFix 1 on both OSes.

SSH Primer

To exploit an SSH server, it is imperative to understand how the SSH protocol works. This post here does a wonderful technical explanation on how the server and client establish an encrypted communication channel using what is called an SSH handshake. The following graphic shows a simplified version of how the handshake works:

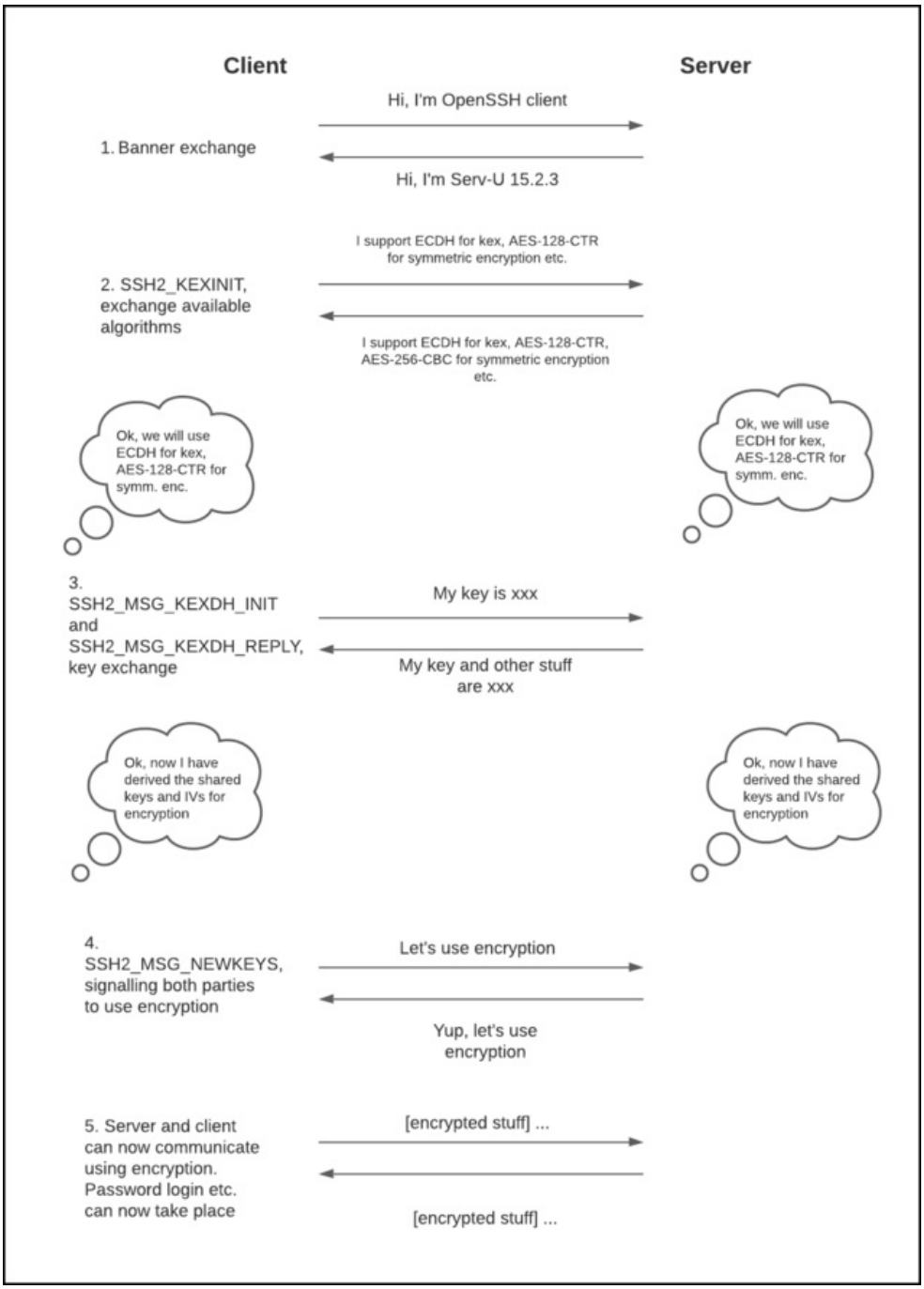


Figure 1. Simplified SSH Handshake

Initially, all packets are sent unencrypted. After exchanging banners, the server and client exchange SSH2_MSG_KEXINIT packets and both parties would decide on what key exchange and symmetric encryption algorithms to use based on a fixed set of rules. The client then sends an SSH2_MSG_KEXDH_INIT packet and the server responds with an SSH2_MSG_KEXDH_REPLY packet to facilitate key exchange. Finally, with both parties sharing a common secret key, the client sends SSH2_MSG_NEWKEYS, which causes both parties to start sending encrypted packets to each other.

Initial analysis

The initial Microsoft article provided me with little to go with; I knew Hotfix 2 patched the bug, and what error Serv-U will log when the exploit fails, as listed below:

```
EXCEPTION: C0000005; CSUSSHSocket::ProcessReceive(); Type: 30; puchPayLoad = 0x03e909f6; nPacketLength = 76; nBytesReceived = 80; n
```

With a little bit of work, I managed to get my hands on both Hotfix 1 and 2 packages. Inside, the main file of interest would be Serv-U.dll, which does the majority of the SSH handling. For my analysis, I would be using the dll from Hotfix 1.

I was quickly able to locate the (16-bit) error string mentioned in the article and associate it with the ProcessReceive function at 0x180144E90.

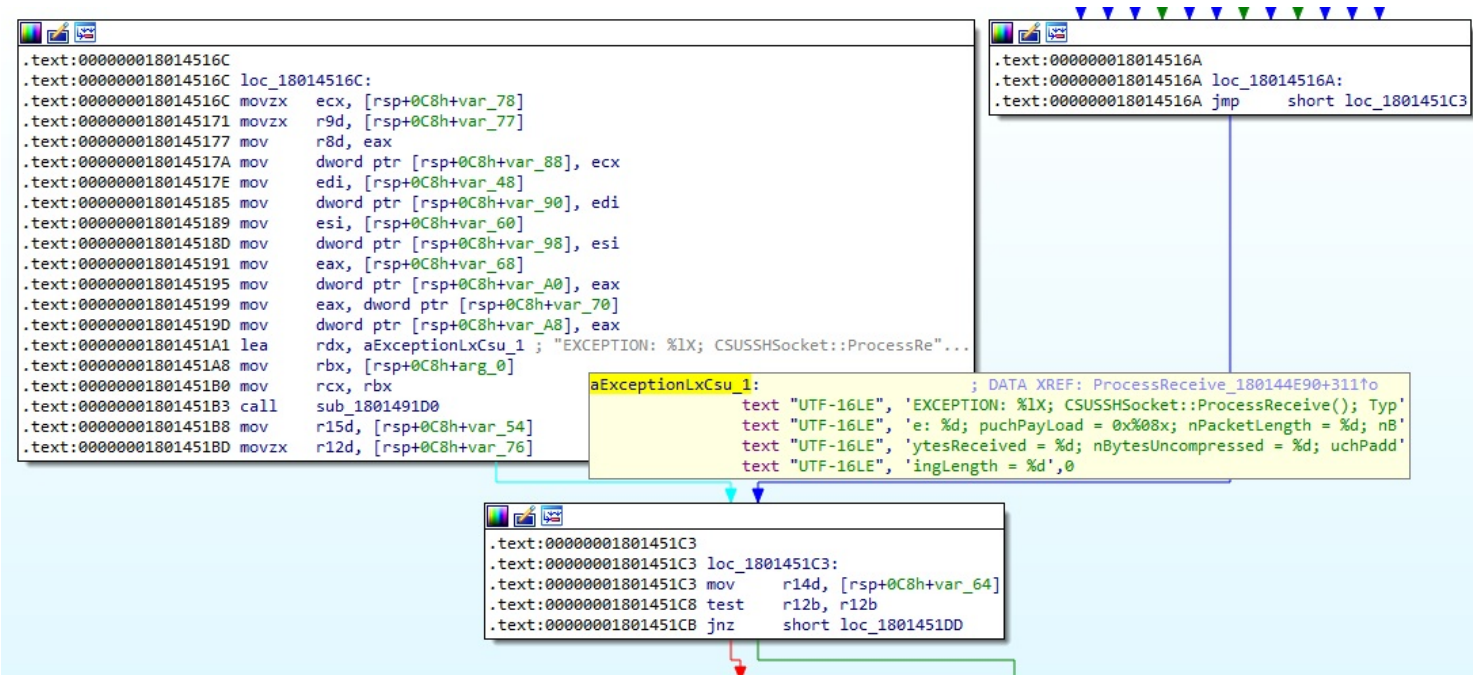


Figure 2. Error string in exception handler

From a cursory analysis of the case switch statement inside ProcessReceive, I was able to deduce that the case switch code read SSH packets and dispatched them to their respective handling functions based on type message (SSH2_MSG_KEXINIT etc.).

```
92        switch ( v22 )
93        {
94          case 2:
95            if ( (*(_BYTE *)(a1 + 400) & 8) != 0 )
96            {
97              v15 = L"SSH_MSG_IGNORE";
98              goto LABEL_14;
99            }
100           break;
101         case 3:
102           if ( (*(_BYTE *)(a1 + 400) & 8) != 0 )
103           {
104             v15 = L"SSH_MSG_UNIMPLEMENTED";
105             goto LABEL_14;
106           }
107           break;
108         case 4:
109           if ( (*(_BYTE *)(a1 + 400) & 8) != 0 )
110           {
111             v15 = L"SSH_MSG_DEBUG";
112 LABEL_14:
113             LODWORD(v20) = v26 + v12 - v29;
114             LODWORD(v19) = v12;
115             sub_1801493C0(a1, *(_QWORD *)(a1 + 280), 32, 18421, v15, v19, v20);
116           }
117           break;
118         case 5:
119           sub_180144C00(a1, (unsigned __int8 *)v25, v26);
120           break;
121         case 20:
122           v16 = v25;
123           if ( !(unsigned int)packet_20_1801445D0(a1, (unsigned __int8 *)v25 - 1, v26 + 1) )
124             goto LABEL_27;
125           break;
126         case 21:
127           packet_21_180144870(a1);
128           break;
129         case 30:
130           packet_30_180144420(a1, (__int64)v25, v26);
131           break;
132         case 32:
00144290 ProcessReceive_180144E90:121 (180144E90)
```

Figure 3. Case switch in ProcessReceive

For this vulnerability, the most important functions would be packet_20_1801445D0, packet_30_180144420 and packet_21_180144870, which correspond to SSH2_MSG_KEXINIT, SSH2_MSG_KEXDH_INIT and SSH2_MSG_NEWKEYS.

Annotating the code

Without symbols, the code was pretty much unreadable, with unknown functions calling more unknown functions. My big break came when I realised that a bunch of functions seemed to share a similar pattern as follows:

```
__int64 __fastcall sub_18015FF40(__int64 a1)
{
  __int64 result; // rax

  if ( (*(_BYTE *)(a1 + 8) & 1) != 0 )
  {
    result = *(_QWORD *)(a1 + 2232);
    if ( result )
      result = ((__int64 (__fastcall *)())result)();
  }
  return result;
}
```

Each function dereferenced a different offset of its first argument and then uses it as a function pointer to call, and this pattern looked awfully like the usage of a virtual table. My theory was confirmed after I found sub_180160D34. Here is an small portion of the function:

```
v1 = 0;
  if ( (*(_BYTE *)(a1 + 8) & 1) != 0 )
    return 1;
  v4 = LoadLibraryW(L"libeay32.DLL");
  *(_QWORD *)(a1 + 16) = v4;
  if ( !v4 )
    goto LABEL_331;
  BIO_new_fp = GetProcAddress(v4, "BIO_new_fp");
  *(_QWORD *)(a1 + 520) = BIO_new_fp;
  if ( !BIO_new_fp )
    return 0;
  BIO_free = GetProcAddress(*(HMODULE *)(a1 + 16), "BIO_free");
  *(_QWORD *)(a1 + 528) = BIO_free;
  if ( !BIO_free )
    return 0;
  BIO_new_socket = GetProcAddress(*(HMODULE *)(a1 + 16), "BIO_new_socket");
  *(_QWORD *)(a1 + 536) = BIO_new_socket;
  if ( !BIO_new_socket )
    return 0;
  BIO_s_mem = GetProcAddress(*(HMODULE *)(a1 + 16), "BIO_s_mem");
  *(_QWORD *)(a1 + 544) = BIO_s_mem;
  if ( !BIO_s_mem )
    return 0;
  BIO_s_file = GetProcAddress(*(HMODULE *)(a1 + 16), "BIO_s_file");
  *(_QWORD *)(a1 + 552) = BIO_s_file;
```

As can be seen, the function looks up OpenSSL functions from libeay32.dll and use them to populate the virtual table at a1. With this understanding, I defined the structure for a1 and wrote an IDAPython script to find all the wrapper functions with the same pattern as sub_18015FF40 and rename them based on their API call. Here is what the code looks like after:

```
v1 = 0;
  if ( (a1->pad[8] & 1) != 0 )
    return 1;
  v4 = LoadLibraryW(L"libeay32.DLL");
  a1->libeay32 = v4;
  if ( !v4 )
    goto LABEL_331;
  BIO_new_fp = GetProcAddress(v4, "BIO_new_fp");
  a1->BIO_new_fp = BIO_new_fp;
```

```
  if ( !BIO_new_fp )
    return 0;
  BIO_free = GetProcAddress(a1->libeay32, "BIO_free");
  a1->BIO_free = BIO_free;
  if ( !BIO_free )
    return 0;
  BIO_new_socket = GetProcAddress(a1->libeay32, "BIO_new_socket");
  a1->BIO_new_socket = BIO_new_socket;
  if ( !BIO_new_socket )
    return 0;
  BIO_s_mem = GetProcAddress(a1->libeay32, "BIO_s_mem");
  a1->BIO_s_mem = BIO_s_mem;
  if ( !BIO_s_mem )
    return 0;
  BIO_s_file = GetProcAddress(a1->libeay32, "BIO_s_file");
  a1->BIO_s_file = BIO_s_file;
```

The same function after renaming:

```
__int64 __fastcall DH_free_18015FF40(struct COpenSSL *a1)
{
  __int64 result; // rax

  if ( (a1->pad[8] & 1) != 0 )
  {
    result = (__int64)a1->DH_free;
    if ( result )
      result = ((__int64 (__fastcall *)())result)();
  }
  return result;
}
```

Now, with my own makeshift “symbols”, understanding the code became a lot easier.

Key Exchange Init (SSH2_MSG_KEXINIT)

When parsing a packet SSH2_MSG_KEXINIT, we eventually reach a function I called parse_1801403C0.

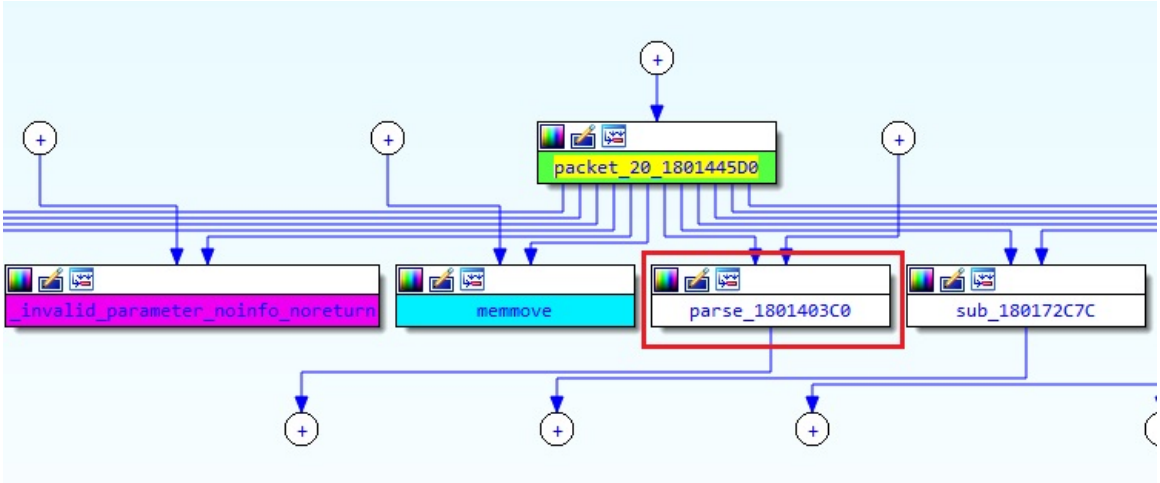


Figure 4. Link between packet_20_1801445D0 and parse_1801403C0

It makes 6 functions calls sequentially, with 2 of them to prep_symm_enc_18013F840:

```
__int64 __fastcall parse_1801403C0(__int64 a1, unsigned __int8 **a2, _DWORD *a3)
{
  unsigned __int8 v6; // r13
  __int64 v7; // rdx
  __int64 v8; // rcx
  int v9; // er9
  int v10; // er8
  int v11; // edx
  __int64 v12; // rax
  int v14; // [rsp+20h] [rbp-48h] BYREF
  char v15[64]; // [rsp+28h] [rbp-40h] BYREF
  char v16; // [rsp+88h] [rbp+20h] BYREF

  v6 = 0;
  CSUString::CSUString((CSUString *)v15);
  if ( sub_18017311C(a2, a3, (__int64)v15, 0) )
  {
    mark_1801406B0(a1, (__int64)v15, (int *)(a1 + 1016));
    if ( sub_18017311C(a2, a3, (__int64)v15, 0) )
    {
      sub_1801408E0(a1, (__int64)v15, (__int64 *)(a1 + 1024));
      if ( sub_18017311C(a2, a3, (__int64)v15, 0) )
      {
        prep_symm_enc_18013F840(a1, (__int64)v15, (__int64 **)(a1 + 1112), 0);
        if ( sub_18017311C(a2, a3, (__int64)v15, 0) )
        {
          prep_symm_enc_18013F840(a1, (__int64)v15, (__int64 **)(a1 + 1120), 1);
          if ( sub_18017311C(a2, a3, (__int64)v15, 0) )
          {
            sub_180140CF0(a1, (__int64)v15, (_QWORD *)(a1 + 1144));
            if ( sub_18017311C(a2, a3, (__int64)v15, 0) )
            {
              sub_180140CF0(a1, (__int64)v15, (_QWORD *)(a1 + 1152));
              if ( *(_QWORD *)(a1 + 1112) )
              {
                if ( *(_QWORD *)(a1 + 1120) )
                {
                  if ( *(_QWORD *)(a1 + 1144) )
                  {
                    if ( *(_QWORD *)(a1 + 1152) )
```

Figure 5. Inside parse_1801403C0

If we look into prep_symm_enc_18013F840, we notice that it involved strings such as “aes128-cbc” and made use of OpenSSL functions such as EVP_aes_128_cbc and EVP_CipherInit_ex (see Figure 6). This pattern of OpenSSL API usage is used for symmetric encryption/decryption, and an example can be found here. It is therefore logical to assume that the calls to prep_symm_enc_18013F840 were to implement the symmetric encryption portion of the SSH handshake.

```
while ( 1 )
{
  v8 = (char *)v58[1];
  v9 = (char *)v58[0];
  if ( v7 >= (unsigned __int64)((v58[1] - v58[0]) >> 4) )
    break;
  v10 = (char *)v58[0] + 16 * v7;
  if ( sub_18015CE20((__int64)v10, (__int64)L"aes128-cbc") && (*(_BYTE *)(a1 + 1172) & 1) != 0 )
  {
    v11 = operator new(0x408ui64);
    if ( v11 )
    {
      v12 = CRhinoProductSocket::GetOpenSSL();
      sub_18013EE80((__int64)v11, v12, a4);
      *(_QWORD *)v11 = &cryptoAES128::`vftable';
      *((_DWORD *)v11 + 4) = 16;
      *((_DWORD *)v11 + 5) = 16;
      *((_QWORD *)v11 + 1) = "aes128-cbc";
      v13 = *((_QWORD *)v11 + 127);
      if ( v13 )
      {
        v14 = EVP_aes_128_cbc_180160958(*((struct COpenSSL **)v11 + 128));
        v15 = EVP_CipherInit_ex_1801606A4(
                *((struct COpenSSL **)v11 + 128),
                *((_QWORD *)v11 + 127),
                v14,
                0,
                0i64,
                0i64,
                *(_DWORD *)(v13 + 16));
        v16 = (void *)*((_QWORD *)v11 + 127);
        if ( v15 )
        {
          sub_180160678(*((struct COpenSSL **)v11 + 128));
        }
```

Figure 6. Inside prep_symm_enc_18013F840

Since there was a pair of calls to the prep_symm_enc_18013F840, I deduced that the calls were responsible for creating OpenSSL EVP contexts for the client-to-server (a1 + 1112) and server-to-client (a1 + 1120) encrypted communication

channels respectively (Note: client-to-server and server-to-client comms can use different encryption algorithms).

Recall that a key exchange has not taken place yet when the server and client are exchanging SSH2_MSG_KEXINIT packets. As such, the server does not possess shared keys and IVs with the client and can only call EVP_EncryptInit_ex with most fields as null, which is valid as stated here:

> EVP_EncryptInit_ex() sets up cipher context **ctx** for encryption with cipher **type** from ENGINE **impl**. **ctx** must be initialized before calling this function. **type** is normally supplied by a function such as EVP_aes_256_cbc(). If **impl** is NULL then the default implementation is used. **key** is the symmetric key to use and **iv** is the IV to use (if necessary), the actual number of bytes used for the key and IV depends on the cipher. It is possible to set all parameters to NULL except **type** in an initial call and supply the remaining parameters in subsequent calls, all of which have **type** set to NULL. This is done when the default cipher parameters are not appropriate.

Figure 7. Documentation for EVP_EncryptInit_ex

Actual Key Exchange (SSH2_MSG_KEXDH_INIT)

Moving on to the parsing of SSH2_MSG_KEXDH_INIT, or packet 30, the bulk of the code logic is in a function I (aptly) named kex_shiz_180142950. Towards the end of the function, we see 6 very similar pieces of code:

```
807  if ( *(_QWORD *)(a1 + 1112) )
808  {
809    v143 = sub_1801418D0(a1, 'A', *(_DWORD *)(a1 + 404), (__int64)Src, v6, a1 + 412, v7[2]);
810    v144 = v143;
811    if ( v143 )
812    {
813      (*(void (__fastcall **)(_QWORD, void *))(**(_QWORD **)(a1 + 1112) + 16i64))(*(_QWORD *)(a1 + 1112), v143);
814      free(v144);
815    }
816  }
817  if ( *(_QWORD *)(a1 + 1120) )
818  {
819    v145 = sub_1801418D0(a1, 'B', *(_DWORD *)(a1 + 404), (__int64)Src, v6, a1 + 412, v7[2]);
820    v146 = v145;
821    if ( v145 )
822    {
823      (*(void (__fastcall **)(_QWORD, void *))(**(_QWORD **)(a1 + 1120) + 16i64))(*(_QWORD *)(a1 + 1120), v145);
824      free(v146);
825    }
826  }
827  if ( *(_QWORD *)(a1 + 1112) )
828  {
829    v147 = sub_1801418D0(a1, 'C', *(_DWORD *)(a1 + 404), (__int64)Src, v6, a1 + 412, v7[2]);
830    v148 = v147;
831    if ( v147 )
832    {
833      (*(void (__fastcall **)(_QWORD, void *))(**(_QWORD **)(a1 + 1112) + 8i64))(*(_QWORD *)(a1 + 1112), v147);
834      free(v148);
835    }
836  }
837  if ( *(_QWORD *)(a1 + 1120) )
838  {
839    v149 = sub_1801418D0(a1, 'D', *(_DWORD *)(a1 + 404), (__int64)Src, v6, a1 + 412, v7[2]);
840    v150 = v149;
841    if ( v149 )
842    {
843      (*(void (__fastcall **)(_QWORD, void *))(**(_QWORD **)(a1 + 1120) + 8i64))(*(_QWORD *)(a1 + 1120), v149);
844      free(v150);
845    }
846  }
```

Figure 8. 6 pieces of code (not all inside image)

The purpose behind this code can be found in the RFC, specifically in section 7.2. Essentially, after the key exchange has occurred, this code is used to generate shared keys and IVs with the client. Note that EVP_EncryptInit_ex is called a second time on a1 + 1120, but this time with the IV and key. With this, encryption / decryption can take place.

Switch to encrypted communication (SSH2_MSG_NEWKEYS)

So far, all packets had been sent in the plain. After receiving the last plaintext packet, SSH2_MSG_NEWKEYS, the server calls calls packet_21_180144870 which assigns a1 + 1120 to a1 + 1104.

```
44  *(_QWORD *)(a1 + 1104) = *(_QWORD *)(a1 + 1120);
45  *(_QWORD *)(a1 + 1120) = 0i64;
46  v5 = *(void (__fastcall ****)(_QWORD, __int64))(a1 + 1128);
47  if ( v5 )
48    (**v5)(v5, 1i64);
49  *(_QWORD *)(a1 + 1128) = *(_QWORD *)(a1 + 1144);
50  *(_QWORD *)(a1 + 1144) = 0i64;

00143C70 packet_21_180144870:1 (180144870)
```

Figure 9. Now a1 + 1104 holds the AES context

From now on, when sending encrypted packets to the client, the server will use this EVP_CIPHER_CTX object stored in a1 + 1104.

Bindiffing

Now, on to the actual bindiffing. The patch was deceptively simple, which made me doubt my judgement quite a couple of times throughout the analysis. As can be seen below, in the patched version, the patch adds a check at the start of packet_30_180144420 to ensure the value of a1 + 408 is 3 before continuing:

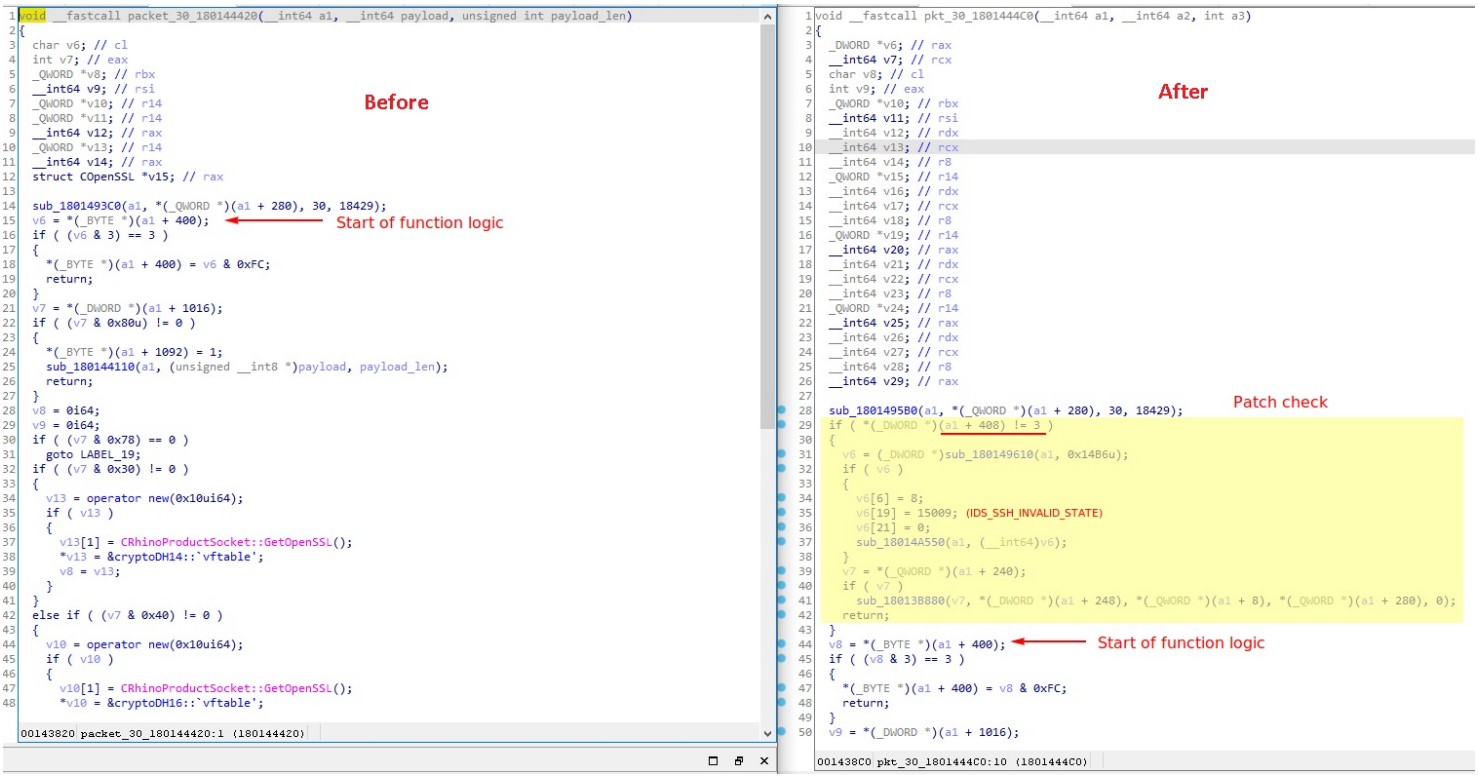


Figure 10. Change to packet_30_180144420

When the updated kex_shiz_180142950 completes successfully, it then assigns 4 to a1 + 408...

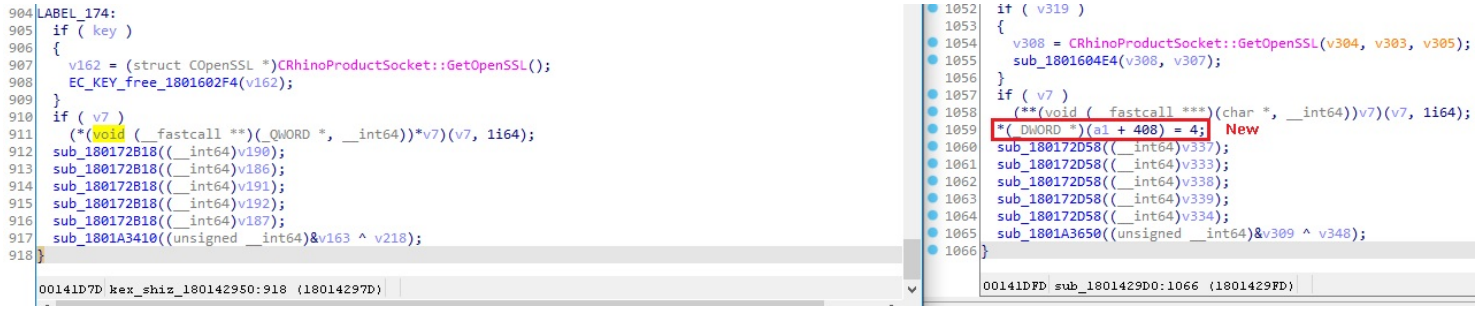


Figure 11. Change to kex_shiz_180142950

...which will be checked for in the updated packet_21_180144870:

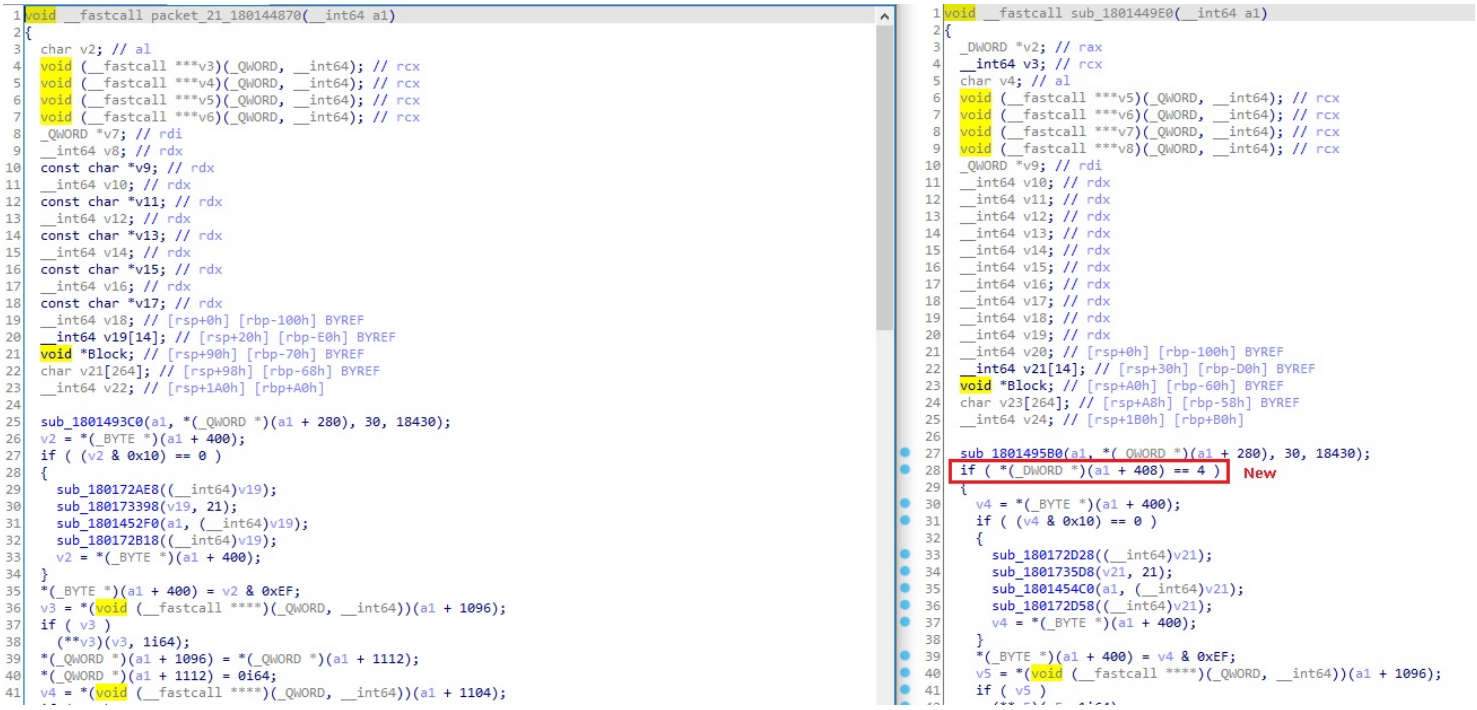


Figure 12. Change to packet_21_180144870

With the patch, the order of packets received is now restricted to the correct order specified by the RFC. For example, if the server receives SSH2_MSG_KEXDH_INIT instead of SSH2_MSG_KEXINIT as the first packet after banner exchange, the function parsing the packet will immediately bail as a1 + 408 is not set to the correct value of 3.

From here, I had 3 hypotheses of how the bug may be triggered:

1. Send SSH2_MSG_KEXDH_INIT as the first packet
2. Send an invalid SSH2_MSG_KEXINIT packet that causes parsing to fail and leave some fields uninitialised, then send an SSH2_MSG_KEXDH_INIT
3. Rearrange SSH2_MSG_KEXINIT, SSH2_MSG_KEXDH_INIT and SSH2_MSG_NEWKEYS in an invalid order

In retrospect, the log entry provided by Microsoft became a red herring in my analysis, as it made me incorrectly assume that the root cause of the bug was somehow in kex_shiz_180142950 which parses SSH2_MSG_KEXDH_INIT. After the first idea was quickly proven wrong, I moved on to the second idea and tried to send all kinds of invalid SSH2_MSG_KEXINIT packets in hopes of triggering a crash, which did not work.

It was only a month of aimlessly testing that I thought of the third possibility. By that time, the Microsoft blog post was released for 1-2 days but I had no idea. After a little testing, I was amazed to find out that just by sending a single SSH2_MSG_NEWKEYS packet, I managed to crash the server. It was that simple all along. However, looking at the logs and crash location, I quickly realised the crash was not what was reported by Microsoft.

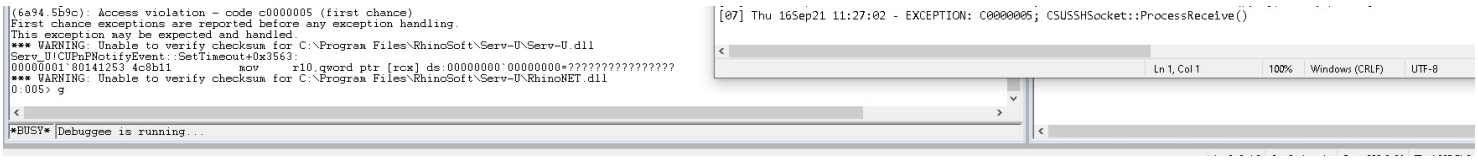


Figure 13. The first crash

The crash happened instead in the packet parsing function at get_pkt_1801411E0, which I did not expect. After some brief analysis, I found the reason: after SSH2_MSG_NEWKEYS is received, the server will treat that all subsequent packets received from the client as encrypted and would try to use the EVP context at a1 + 1096 to decrypt any subsequent packets received; however, without a prior SSH2_MSG_KEXINIT, the field a1 + 1096 holding the decryption context would be left as NULL, causing a null-pointer dereference.

```
char __fastcall get_pkt_1801411E0(__int64 a1, unsigned __int8 *a2, const void **a3, unsigned int *a4, int *a5, _DWORD *a6, int *a7, _BYTE *a8, char *a9, char **a10)
{
  __int64 v14; // rax
  unsigned int v15; // esi
  __int64 v16; // r12
  void *v17; // rax
  unsigned int v18; // er13
  char *v19; // rdx
  unsigned int v20; // edi
  __int64 v21; // rax
  char *v22; // rax
  int v23; // eax
  int v24; // esi
  const void *v25; // rax
  char *v26; // rcx
  int v27; // esi
  int v28; // er13
  int v30; // [rsp+40h] [rbp-20h] BYREF
  unsigned int v31; // [rsp+44h] [rbp-1Ch] BYREF
  char *v32; // [rsp+48h] [rbp-18h] BYREF
  char *v33[2]; // [rsp+50h] [rbp-10h] BYREF
  char v34; // [rsp+A0h] [rbp+40h]
  unsigned __int8 *v35; // [rsp+A8h] [rbp+48h]

  v35 = a2;
  v14 = *(_QWORD *)(a1 + 0x448);
  if ( v14 )
    v15 = *(_DWORD *)(v14 + 16);
  else
    v15 = 8;
  v16 = a1 + 696;
  if ( !(unsigned int)sub_180166764(a1 + 696) && *a4 >= v15 )
  {
    v17 = (void *)sub_1801663DC(v16, v15, 0);
    if ( *(_BYTE *)(a1 + 1180) )
      (*(void (__fastcall **)(_QWORD, void *, const void *, _QWORD))(**(_QWORD **)(a1 + 1096)
                                                                    + 32i64))(// crash point  NULL
        *(_QWORD *)(a1 + 1096),
        v17,
        *a3,
        v15);
    else
      memcpy(v17, *a3, v15);
    *a3 = (char *)*a3 + v15;
    *a4 -= v15;
    *a2 = 1;
  }
```

Figure 14. Cause of crash

To remedy this issue was simple: I just had to tell the server that client-to-server packets could only be sent in the plain. To do this, I sent an SSH2_MSG_KEXINIT packet before SSH2_MSG_NEWKEYS specifying that client-to-server encryption could only be “none”, which was supported (though not encouraged) according to the RFC. After this, I sent an SSH2_MSG_KEXDH_INIT packet and boom, the server crashed, with the same log entry as Microsoft documented.

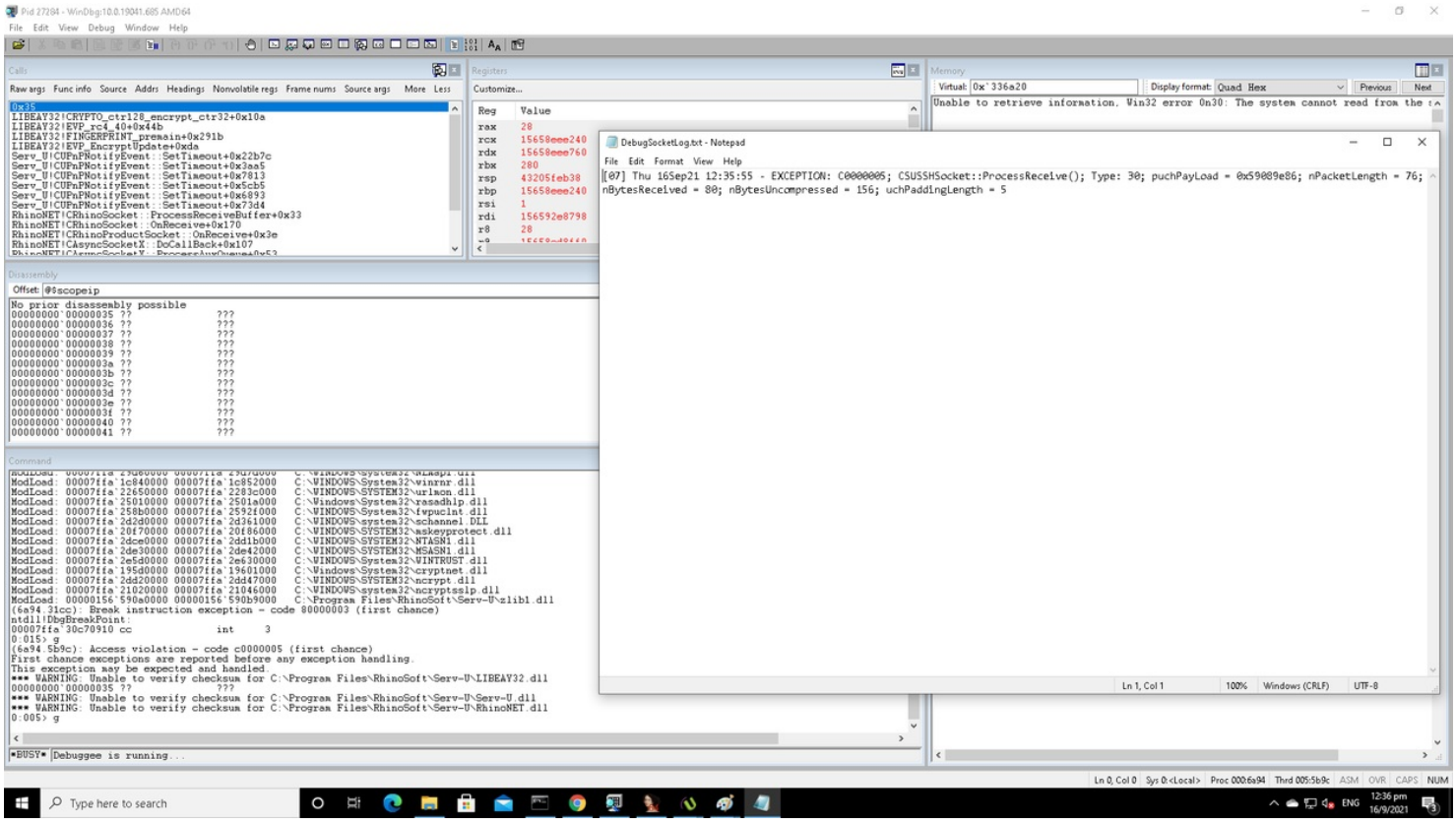


Figure 15. Crash reproduced!

The following is my analysis of the root cause of the vulnerability: as the server did not receive an SSH2_MSG_KEXDH_INIT packet, no key exchange took place and the previously mentioned second call to EVP_EncryptInit_ex would hence not occur.

However, as the server does not check if an SSH2_MSG_KEXDH_INIT packet had been received before using new keys, it accepts the subsequent SSH2_MSG_NEWKEYS packet as valid and treats the encryption context at a1 + 1120 as valid.

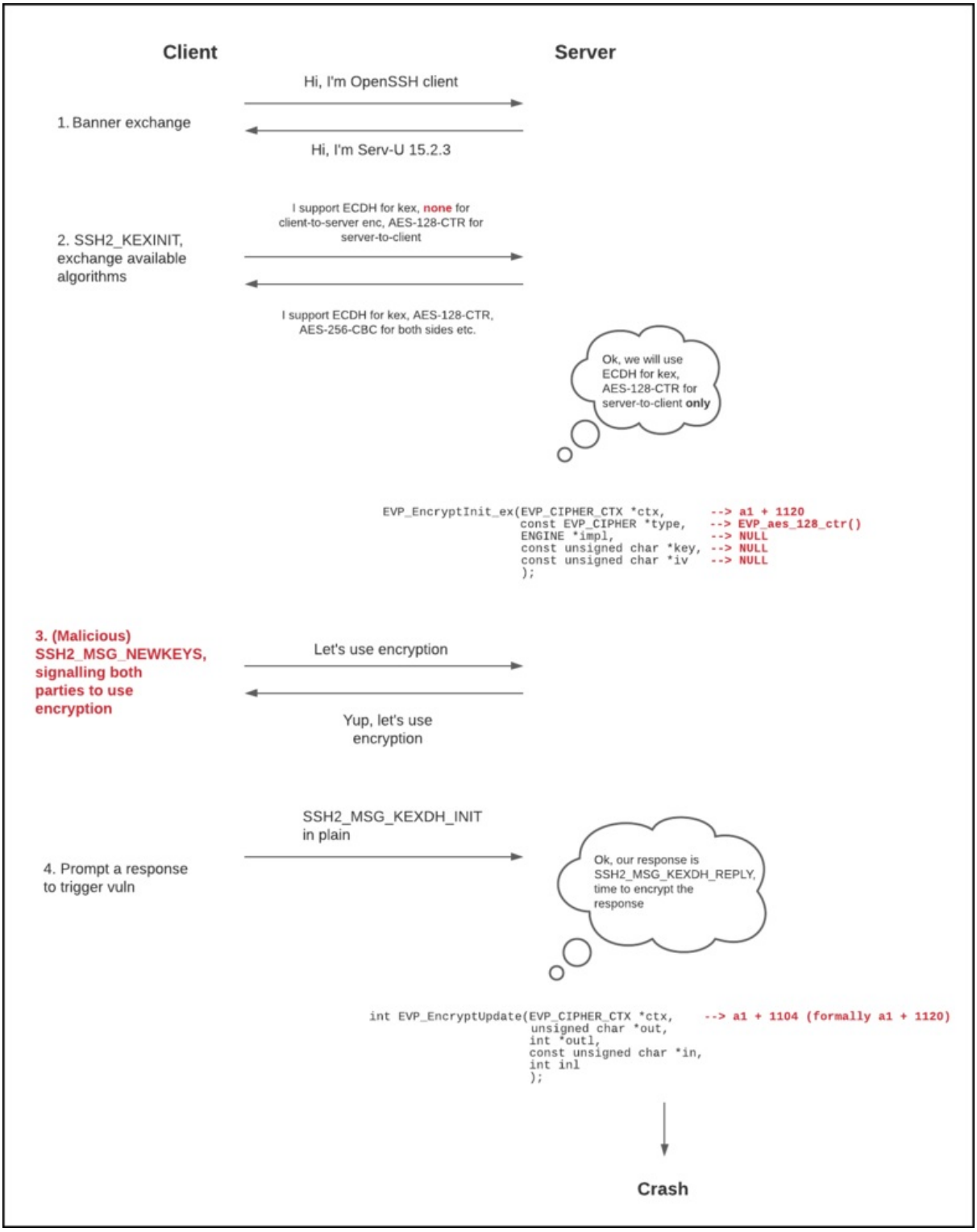


Figure 16. Process of triggering the vulnerability

When I then send an SSH2_MSG_KEXDH_INIT packet, or in fact any packet that elicits a server-side response, the server will attempt to encrypt its response packet, and it does this with EVP_EncryptUpdate. When the EVP_CIPHER_CTX context is initialised with AES 128 CTR, the EVP_EncryptUpdate call eventually leads to a call to aes_ctr_cipher:

```
static int aes_ctr_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out,
                          const unsigned char *in, size_t len)
{
    int n = EVP_CIPHER_CTX_get_num(ctx);
    unsigned int num;
    EVP_AES_KEY *dat = EVP_C_DATA(EVP_AES_KEY,ctx);

    if (n < 0)
        return 0;
    num = (unsigned int)n;

    if (dat->stream.ctr)
        CRYPTO_ctr128_encrypt_ctr32(in, out, len, &dat->ks,
                                    ctx->iv,
```

```
                                EVP_CIPHER_CTX_buf_noconst(ctx),
                                &num, dat->stream.ctr);
    else
        CRYPTO_ctr128_encrypt(in, out, len, &dat->ks,
                              ctx->iv,
                              EVP_CIPHER_CTX_buf_noconst(ctx), &num,
                              dat->block);
    EVP_CIPHER_CTX_set_num(ctx, num);
    return 1;
}
```

In the else case, dat->block, or ctx->cipher_data->block, is used as a function pointer. ctx->cipher_block comes from a OPENSSL_zalloc here. The astute reader might be wondering how OPENSSL_zalloc can cause uninitialised memory use when it zeroes out its allocated heap buffer.

As it turns out, doing a git blame on line 294 shows us that the commit which introduced the use of OPENSSL_zalloc was made in 2006. Unfortunately, Serv-U appears to use a version of libeay32.dll that was dated back to 2005. If only they had just used a more up-to-date libeay32, this vulnerability would probably had been unexploitable.

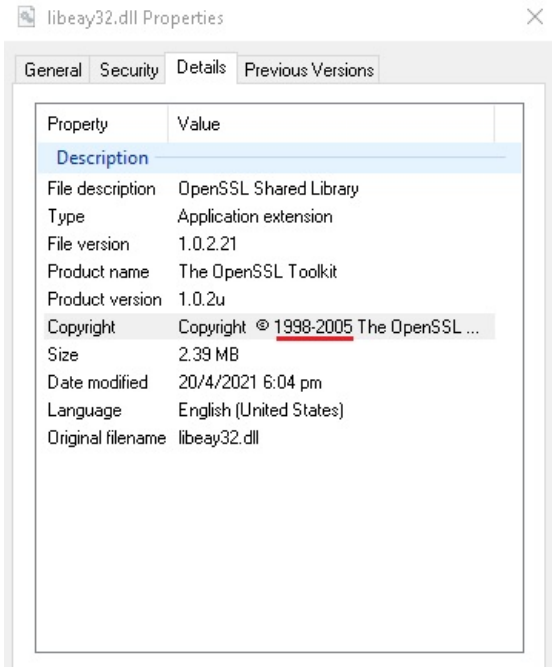


Figure 17. libeay32.dll used by Serv-U

Moving on to cipher_data->block, we see that it is assigned in aes_init_key, which is called from here. As seen from the line 369, if we do not call EVP_CipherInit_ex with a non-NULL key, this chain of calls will never happen, leaving cipher_data->block uninitialised. This causes the use of an uninitialised function pointer.

Conclusion

Where do we go from here? Stay tuned for part 2 on a detailed writeup on how I exploited this vulnerability to gain RCE.
